Machine Learning in Compilers

Hugh Leather

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2011

Abstract

Designing a compiler so that it produces optimised code is a difficult task because modern processors are complex. Compiler writers need to spend months to finely tune a heuristic for any architecture, but whenever a new processor comes out the compiler's heuristics will need to be retuned for it. This is, typically, too much effort and so, in fact, most compilers are out of date. Machine learning has been shown to help, creating tools that can predict how best to compile new programs from observations made about programs compiled in the past. Many hurdles still remain, however, and while experts no longer have to worry about the details of heuristic parameters, they must focus on the details of the machine learning process instead.

This thesis develops techniques so that, where human compiler experts were needed to manage the machine learning experiments before, they are required no longer; paving the way for a completely automatic, retuning compiler.

First, we tackle the most outstanding area requiring human involvement; feature generation. In all previous machine learning works for compilers, the features, which describe the important aspects of each example to the machine learning tools, must be constructed by an expert. Should that expert choose features poorly, they will miss crucial information without which the machine learning algorithm can never excel. We show that not only can we automatically derive good features, but that these features outperform those of human experts. We demonstrate our approach on loop unrolling, and find we do better than previous work, obtaining 76% of the available performance, more than the 59% of previous state of the art.

Next, we demonstrate a new method to efficiently capture the raw data needed for machine learning tasks. The iterative compilation on which machine learning in compilers depends is typically time consuming, often requiring months of compute time. The underlying processes are also noisy, so that most prior works fall into two categories; those which attempt to gather clean data by executing a large number of times and those which ignore the statistical validity of their data to keep experiment times feasible. Our approach, however, guarantees clean data while adapting to the experiment at hand, needing an order of magnitude less work than prior techniques.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own, except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Hugh Leather and Michael O'Boyle and Bruce Worton. "Raced Profiles: Efficient Selection of Competing Compiler Optimizations". In *Proceedings of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), 25-28 June 2009 Dublin, Ireland

- Hugh Leather and Edwin Bonilla and Michael O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO), March 2009 Seattle, United States of America

(Hugh Leather)

Table of Contents

Chapter 1

Introduction

Modern processors have become increasingly complex; they have grown in terms of the number of transistors devoted to different functional units, memories and techniques designed to speed up the execution of programs. As this complexity has arisen, the interactions between all of these various components and the optimisations available in the compiler have made it hard to predict how the whole computer system will react to changes in the compiler's heuristics.

What is worse is that even if a human is able to understand enough of the interdependencies in the computer and can afford the months of time needed to properly tune the compilers heuristics, his work will apply to only one version of the architecture. Even different processors from the same processor family require different heuristic tunings, if the job is to be done well. New architectures are constantly being developed, each bringing new and daunting challenges to the compiler writer. Even if the architecture is fixed, a change, upstream to the compiler, will invalidate the tuning already done to those phases which come after it. Humans cannot keep up and many of today's compilers cope by simply accepting the inevitable, remaining permanently out of date.

Of course, compiler researchers have not stood idly by watching the situation get away from them. They have developed techniques to automate the proper selection of parameters for their compiler's heuristics(Bernstein et al., 1989). First they employed iterative compilation to find the best parameter values, eschewing the compiler's own choices and exploring the space of options afresh each time a new program needed to be compiled(Cooper et al., 2005). The technique worked regardless of changes to the architecture or compiler. The search time, however, was prohibitive and so apart from heavily used libraries and mission critical application kernels, it seemed unlikely to

become widely adopted. The results, however, were impressive and the gains were too good to be given up on; an alternative was needed. Researchers began to experiment with machine learning, allowing a computer to build models which could predict the right values for compiler heuristics based on previous experiments it had conducted. A fully automated, self tuning compiler seemed within reach (Stephenson and Amarasinghe, 2005) and in early experiments the machine learned models performed almost as well as the iterative compilation searches, just without needing any search time from the users of the updated compiler.

Unfortunately, machine learning for compilers still requires the involvement of human experts in many stages of the design and without due care and attention, the learning will be either inadequate or inefficient. It is quite possible for the researcher to provide too little or irrelevant information to the machine learner so that its ability to predict the right heuristic values is compromised. It is also quite possible that the data to be learned over are poorly or wastefully produced. The compiler writer needs to be insulated from these issues, to have them managed on his behalf.

1.1 The Problem

Machine learning for compilers operates as follows. First a heuristic to be replaced is identified. Then a number of benchmarks are compiled with different values for the heuristic parameters. Each of these compiled program versions is then run and timed to find out which is the fastest; because of noise in the measurement, each version must be run several times to be sure to produce statistically valid results. Next, the researcher considers what information would be useful if he were deciding the right heuristic value instead of the machine and he implements code to extract these summaries (called features in machine learning parlance (Kohavi and John, 1997)). Now, for each example benchmark he has features describing it and knows the best possible value for the heuristic. Armed with these data (called training data) he asks a machine learning tool to learn a predictive model which, when given the features describing a new, unseen program, will guess a good heuristic value to use. He replaces the original compiler's heuristic with the features implementation and the predictive model and, if he has done the job well, the compiler will perform better than it did before his intervention. Whenever the heuristic must be retuned, either because of a change to the architecture or due to some other change elsewhere in the compiler, the compiler writer repeats the automated process, regenerating the training data, learning a new

model and replacing the heuristic; all mechanically and without undue effort.

The smooth running of that process is predicated on the researcher succeeding at two things; he must choose the right features to summarise the programs and he must measure the execution times of the compiled programs both correctly and efficiently. The following two sections describe the problems inherent in these tasks in more detail.

1.1.1 Feature Generation

No machine learning tool will be able to learn a good predictive model for an heuristic if the features describing the training programs are poor. If the compiler writer chooses to describe programs with features like "the age of the programmer" or "the amount of white-space in each function", then it would be quite surprising if a machine learning tool could make good progress. Although these examples might seem unlikely to occur in practice, the interactions between the features and the machine learning algorithm are, in fact, quite complex. Features that are based on human intuition may not be the best features to choose. Features may not represent all of the relationship between the program and the desired outcome or, even if they do, they may not work sufficiently well with the chosen machine learning algorithm.

For the compiler writer, the difficulty of the situation is exacerbated by the limitless number of choices open to him for possible features. The data structures that make up the compiler's internal representations are trees and graphs; multi-dimensional sequences are common with extraneous information appended from the result of various analyses. To start with, the compiler writer will typically take whatever data the previous, human-created heuristic used, and then add to this some histogram of the instructions in each basic block or count each type of node in the abstract syntax tree. This, however, is only the first level of possibilities; one might count the number of nodes of one type whose first child is of another and so on, indefinitely; many different aggregation functions can be used, and any combination of arithmetic and logic can filter and transform the features at will. As this thesis will show, there are an infinite number of ways to summarise the data.

Until now, the compiler writer has taken the very sensible approach of implementing those features that appeal to his intuition or that are easy to extract. He has embarked on further investigation only when the results fall short of his expectations. He does not know if he has the best possible features for his compiler; he knows only that he has outdone what went before and leaves it at that. His intuition, as this thesis will

show, may well be wrong and he may have lost a good deal of the potential.

What is needed, instead, is a way for the compiler writer to describe and search the infinite variety of features that might occur to him if only his time to explore them were not limited. Let the machine search for good features for him. This thesis will develop just such a technique. The compiler writer need never involve himself again in the quality of his features; the machine will take on that burden for him.

1.1.2 Data Generation

Poor choice of features is not the only thing that can derail a machine learning experiment in compilers. Before anything can be learned, a set of benchmarks must be compiled in different ways with iterative compilation and each then run to find which way is best for each benchmark. This process builds a corpus of training data that the machine learning algorithm will study to build its predictive model. The bigger this corpus, the better. Iterative compilation is expensive, though. An experimenter might spend months of compute time on it and would like to make sure that he runs each program as few times as possible and wastes no unnecessary effort. Indeed, the upfront cost of generating training data is a common cause for complaint by those considering adopting machine learning techniques into production compilers.

Moreover, when measurements of a program's execution time are made there is noise in the signal; computers are affected by so many different factors. There is never only one process contending for resources and the initial starting conditions, down to the state of the cache and the file system will alter the precise timing of a program. Even the temperature plays its part. The compiler writer has to be careful to generate good data or he may end up trying to learn that noise, certainly leading to suboptimal results. He must run each program version enough times so that he can estimate the true execution time to the proper degree. The number of times to repeat the execution depends not on some global, unchanging factor, but rather it is different for each architecture he runs on, for each granularity of experiment and for each program version. To ensure good data, most researchers run each program a large and constant number of times, oblivious to whether that is strictly necessary; a few others run each program a small number of times and quite likely receive poor data as a result, perhaps invalidating their own conclusions.

What is needed, then, is a way to manage the gathering of the training data. To ensure that the results will be statistically significant (or the experiment might just as

well not be done at all) but that at the same time no effort is wasted. Once a program version can already be shown to have no hope of being the best, we should spend no resources on it; only those which might be the best should be given the opportunity to prove it. We need a system that adapts the iterative compilation process to efficiently and correctly generate the data. This thesis presents just such an algorithm.

1.2 Contributions

This thesis presents new techniques that aim to remove the need for human involvement when replacing compiler heuristics with machine learned versions. The previous sections have shown the problems that must be overcome by any compiler writer wishing to use machine learning in their software. These problems are solved by the work in this thesis.

A method to automatically *generate* features and to search for good ones is shown; the compiler writer need not try manually select the right features again. The thesis shows how infinite families of features can be defined; the space can contain all manner of features, both complex and simple, with all variations accounted for. The computer will navigate the space, choosing only those features which are useful. Specifically, this part of thesis:

- Develops a grammatical system to describe feature languages (chapter 5).
- Demonstrates how such a feature family can be searched to find the best possible features for a given compiler heuristic and machine learning algorithm (chapter 6).
- Shows that automatically generating features not only absolves the compiler writer of that effort but also outperforms human derived, expert selected features (section 6.6).

No work about machine learning in compilers has considered searching over the feature space; this thesis is the first to do so.

The thesis also brings a new method to manage the iterative compilation needed to generate the training data for machine learning (chapter 7). The new technique is an order of magnitude more efficient than previous approaches. The new method:

- Determines a subset of the optimisation settings or program versions which are 'better' than all others, to some user supplied significance level.

- Minimises the number of runs required, dropping poorly performing versions early and finishing when sufficient data have been gathered for a decision.
- Provides statistically rigorous results.
- Allows more points in the compiler optimisation space to be examined.

The thesis also presents a complete machine learning compilation system for GCC (chapter 4) and develops a hybrid evolutionary search technique that combines the best of grammatical evolution and genetic programming (section 5.5).

This section has laid out the main contributions of the thesis. The next section describes the overall structure of the thesis.

1.3 Structure

The remainder of this thesis is organized as follows:

Chapter 2 covers the machine learning techniques on which this thesis builds.

Chapter 3 discusses work related to this thesis. The chapter begins with works that search the space of compiler heuristic parameters, either on a per-program basis or to completely replace heuristics. Then the ways that features have been used and how others have fine tuned their feature choices are explored. Finally comes work that advocates statistical rigour in compiler performance measurement and mathematical methods for smart sampling.

Chapter 4 introduces a machine learning compiler, based on GCC, which allows the very fine grained control necessary to support the needs of machine learning. It was used in all of the experiments in this thesis.

Chapter 5 shows that the features used for machine learning in compilers are derived from an infinite space and explains how a family of features can be specified by use of a probabilistic context free grammar. The chapter discusses the design issues that must be taken into account for grammar construction and also covers the low-level operations that support searching over feature grammars.

Chapter 6 builds on the concepts presented in the previous chapter, developing a complete system to search for the best features. It shows how feature generation outperforms human designed features. The work of this chapter is based on that published in (Leather et al., 2009a).

Chapter 7 presents an algorithm for managing the iterative compilation process that generates training data for machine learning. The algorithm ensures that statisti-

cally valid data are acquired with the smallest possible effort expended. Prior works either ignored the statistical significance of their data or wasted an unnecessary amount of time executing programs. The work of this chapter is based on that published in (Leather et al., 2009b).

Chapter 8 concludes the thesis, drawing together the main contributions and discussing future work to be done.

Appendix A provides considerably more detail on the extensible compiler of chapter 4.

1.4 Summary

This chapter has introduced the thesis, explained the problems that the techniques developed in the thesis will solve and described the contributions offered by this thesis. A summary of the structure of the document has also been presented. The next chapter will briefly discuss the concepts used throughout this thesis.

Chapter 2

Background

2.1 Introduction

This chapter gives an overview of the techniques made use of in this thesis. First, common terminology is introduced in section 2.2. Then, section 2.3 describes the most typical supervised machine learning for compilers experiment, to give full context. The specific machine learning algorithms employed in this thesis are presented in section 2.4 and then the evolutionary search approaches that this thesis builds upon are explained in section 2.5. Finally, in section 2.6, several relevant topics from statistics are covered.

2.2 Terminology

The term *machine learning* includes techniques that attempt to have computers learn. *Supervised machine learning* involves learning a function which generalises data presented as a set of training examples. Given a set of pairs, $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$, each of which being an example of some function $\mathbf{y} = f(\mathbf{x}, \mathbf{z})$, a supervised machine learning tool will generate a function $\mathbf{y} = f'(\mathbf{x})$ that is a best guess at replicating the original function f. The new function f' is called a *predictive model*, or just *model*, since it tries to predict the output of the original function for a new input vector that was not present in the training examples. The examples, most usefully, come from some real world observations of random variables. The original function is typically not precisely known and may have additional parameters, $\mathbf{z}$, that are not observed, limiting the ability of the machine learning tool to create a good model. The input vector, $\mathbf{x}$, is called a *feature* vector and the output vector, $\mathbf{y}$, is the *response*.

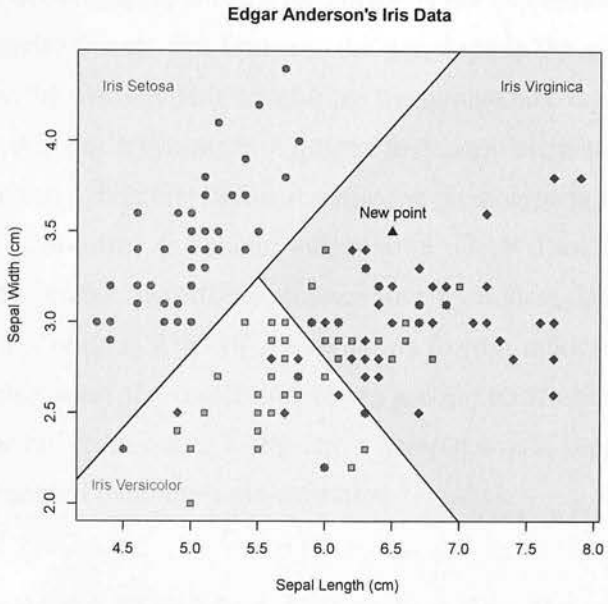


Figure 2.1: An example classification problem. Classification for three different species of iris, Iris setosa (red circles), Iris versicolor (green squares) and Iris virginica (blue diamonds), some overlap. Sepal measurements, for example flowers, are shown. The classification problem is to decide the correct species given measurements for a new flower. A linear classification is shown which performs well for setosa, but poorly for the others. The new point (black triangle) would be classified as virginica.

Classification refers to supervised machine learning techniques for which the response comes from a discreet set. For example, one of the standard classification benchmarks is Anderson's iris data which contains measurements of petal and sepal lengths and widths for three different species of iris. A plot is shown, using two of the features in figure 2.1. The goal is to predict the species of a new plant given the measurements of its petals and sepals. The alternative to classification is *regression* where the response is a continuous real valued number.

Supervised machine learning algorithms may suffer from *over-fitting*. This occurs when the learned model is excessively close to the examples provided for training, while being unable to generalise well to make predictions for new values. As a simple example, consider that a model that is merely a look-up table of the training examples. This model will perfectly predict the answer for the training data but will not predict for new data.

In evolutionary search and genetic algorithms, a *genotype* is the set of data describ-

ing an individual in the search, while a *phenotype* is the expression of the individual's genotype in its environment. For humans, the genotype is the genes of the chromosomes in the cells; the phenotype is set of traits the human has, e.g. hair colour, height, good at running, etc. For a synthetic organism in genetic algorithms, the genotype is the binary or structured data that define it while the phenotype is how it behaves in its test environment. An *intron* is genetic information which does not contribute to the phenotype; several genetic algorithms produce such redundant data.

In statistics, a *sample* is a set of *observation*s from a random variable. Thus, if the random variable were the result of flipping a coin, each observation would be a particular head or tail from a single flip and a sample would be a collection of such heads and tails recorded from multiple coin flips.

2.3 Supervised Machine Learning for Compilers

The majority of machine learning experiments for compilers have followed a typical pattern which is very briefly described in this section.

In the first step, the compiler writer will choose a heuristic to replace with an automatically learned version. For example, he may decide to replace the priority function of the instruction scheduling optimisation with a more accurate, machine learned version. Since supervised machine learning entails building a predictor that best matches empirical results, the compiler writer needs to find out what the desired value of the priority function should be for several examples. This is done by *iterative compilation* (Agakov et al., 2006).

In iterative compilation, as shown in figure 2.2, the compiler writer augments the compiler so that he can force particular heuristic values for different programs. The heuristic values will typically come from some fixed dimensional space and he can build a search routine that will traverse this space. Most often, the interest lies in improving execution time, so the compiler will generate instrumented code that will allow the program to be accurately timed after averaging the results of a few runs. On occasion, however, other performance metrics have been used; code size, energy or some combination thereof (Cooper et al., 1999). The machine used to execute the programs might also be virtual, in the form of a simulator. The end product of the search will be the best found heuristic values for the given program and for machine learning purposes, those values will be gathered for many example programs.

After the data gathering phase of iterative compilation, the compiler writer needs

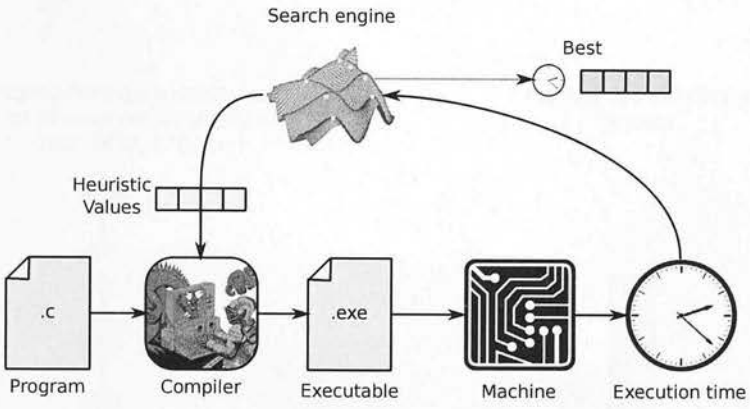


Figure 2.2: An overview of a typical iterative compilation system. A compiler produces instrumented executables from source programs and heuristic values are supplied by a search algorithm. The search attempts to find the heuristics which produce the fastest version of the program on a given machine.

to summarise the programs in preparation for passing that information to a machine learning tool, see figure 2.3. At nearly all points during the compilation of a program the compiler's internal representation of that program will consist of graphs, trees and the complex results of one analysis or another. Machine learning tools generally demand a fixed length vector of numbers to learn from, so these intricate data structures are not suitable. The compiler writer decides what information will be useful for predicting a good value for the heuristic. Typically, this will involve asking what data the compiler writer himself would need, seeing what is used by the existing heuristic or just gathering whatever is easily computable. The elements of the summary vector are called *features*.

The compiler writer now computes the feature vector for each of his benchmarks and pairs these with the best heuristic values he found through iterative compilation, as shown in figure 2.4. These pairs form the *training data* which are passed to the chosen machine learning tool so that it can create a predictive model.

The process of learning a model is exemplified by figure 2.5. Here a number of example points are shown, each with its features and desired heuristic value. The machine learning algorithm fits a curve to the data; the curve represents the model. The curve can then be used to predict a good heuristic value for an unseen point; the prediction is the value of the curve at the point of the new program's features. Feature vectors are typically multi-dimensional and example training data will contain complex patterns, so the 'curve' fitting will be, in fact, a much more involved process.

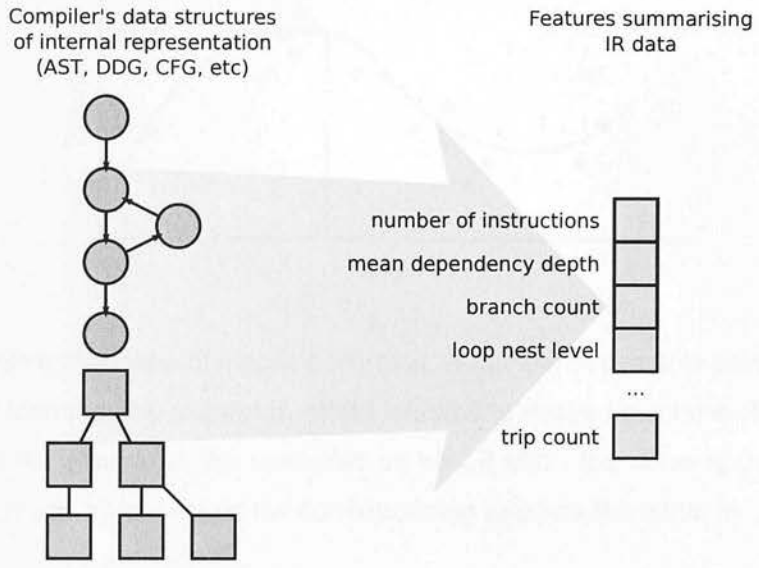


Figure 2.3: Summarising programs into features. The compiler writer decides what information will be most useful to represent the programs and packs them into a finite feature vector.

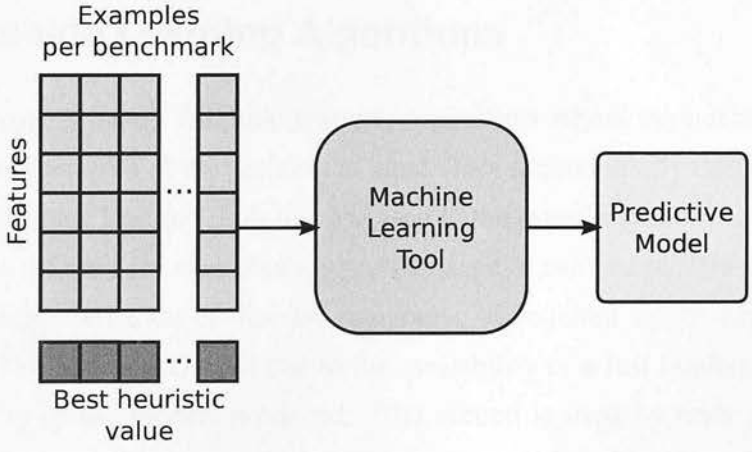


Figure 2.4: Learning a predictive model. The compiler writer gathers a number of examples, through iterative compilation, of the best heuristic values for his benchmarks. These are given to a machine learning tool to create a model.

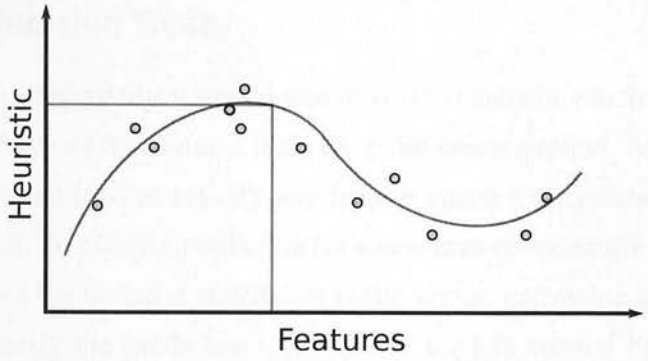


Figure 2.5: Simplified view of machine learning. A number of example points are made available for learning; the examples, relate features to desired outcome. The machine learning tool fits a curve to the examples as best it can - the curve is the model. A prediction is made by looking up the corresponding value on the curve for a new point.

For classification problems, the model describes which portions of the feature space belong to each class, as shown in figure 2.1. Once the model has been learned, it can be inserted into the compiler, and the previous heuristic will then be replaced, as required.

This section has described the typical phases of a machine learning experiment for compilers. The specific machine learning algorithms used in this thesis are covered in the next section.

2.4 Machine Learning Algorithms

There are many different machine learning algorithms whose applicability depends upon the characteristics of the problem at hand. This section briefly describes the two supervised machine learning tools that are used in the experiments of this thesis. Both of these tools are used for classification problems, i.e. those that require the predictive model to select from a set of discreet categories, as required by the experiments of this thesis. The first was chosen due to the availability of a fast implementation and the readability of the models produced. The second is used by prior work and so was chosen for reasons of comparison. This section also explains how the quality of machine learning algorithms is validated.

2.4.1 C4.5 Decision Trees

A decision tree is, essentially, a nested tree of *if* statements in which each conditional tests the value of one of the features from the point being queried. At each leaf of the tree is decision about how to classify any feature vector that matches the conditions leading to that leaf. To obtain a prediction for a new feature vector the tree is evaluated; one simply applies the nested if statements to the vector, narrowing to the appropriate leaf that will specify the prediction. Unlike the models created by many machine learning tools, decision trees are both easy to evaluate and to understand which has made decision trees quite popular.

C4.5 (Quinlan, 1993) builds decision trees so that the conditional of each node splits the feature which is most useful at classifying the examples from the training data. The examples are then partitioned according to that feature and the child nodes are created by the same technique recursively for each partition. To determine which feature is most useful, C4.5 uses the metric of *information gain*, which describes how well the feature separates the examples in relation to their classifications.

The definition of information gain depends upon the concept of *entropy* which quantifies the amount of information in bits needed to describe a set. If a set of examples all have the same classification then it can be compactly described and thus has a low entropy; conversely a random mixture needs more information and has high entropy. For a classification problem, the entropy of a set, S, which consists of c categories for which the proportion of each in the set is p_i, is given in equation 2.1.

$$entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2.1}$$

Information gain is then the expected reduction in entropy that would result from partitioning the set according to a particular feature, as shown in equation 2.2. Here, if F is a feature, $values(F)$ is the set of values that a feature can take. The function $gain(S,F)$, then gives the amount of bits saved in the description by knowing the value of a given feature.

$$gain(S,F) \equiv entropy(S) - \sum_{v \in values(F)} \frac{|S_v|}{|S|} entropy(S_v) \tag{2.2}$$

The C4.5 decision tree algorithm recursively chooses for each node the feature with the greatest value of $gain(S,F)$. Additionally, it copes with missing feature values, handles both discreet and continuous features and removes branches that do not

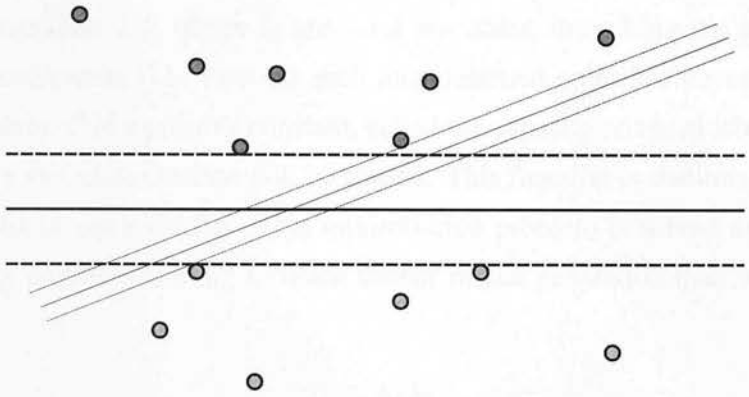


Figure 2.6: Linearly separating examples with a maximum margin hyper-plane. Examples of two different classifications are separated by two candidate hyperplanes. The first (shown with a finer line) separates the classes by a smaller margin than the other. SVMs choose planes which separate by the largest margin. Points touching the margin are called *support vectors*.

significantly contribute, replacing them with leaf nodes and simplifying the tree. The runtime for learning a model with C4.5 is $O(DN(\lg N)^2)$, where D is the number of attributes and N is the number of instances (Bradford et al., 1999).

2.4.2 Support Vector Machine

A linear support vector machine (SVM) (Schlkopf and Smola, 2001) uses a hyperplane to linearly separate the training examples for binary classification. Linear SVMs attempt to find hyperplanes which separate the data by the maximum margin, as shown by figure 2.6. The plane represents the model's guess about the true separation of the data. One can then form a prediction for a new feature vector by asking whether the new point lies on one side of the plane or the other. Thus, if plane is described by $\mathbf{w}.\mathbf{x}+b=0$, then the decision function is given by equation 2.3. In that equation, $f(\mathbf{x})$ returns 1 or -1 indicating the predicted classification.

$$f(\mathbf{x}) = sign(\mathbf{w}.\mathbf{x}+b) \tag{2.3}$$

Linear SVMs need to be able to cope when training data is not perfectly separable. To do this, they will attempt to find the plane which maximally separates as many examples as possible, applying a penalty for any points it fails to separate. Given m training examples, $\langle \mathbf{x}_i, y_i \rangle$ with $y_i \in -1, 1$, for $i \in [1, m]$, the SVM minimises the

function in equation 2.4, where ξ_i are *slack variables*, describing the magnitude of each mis-classification (i.e. how far each mis-classified point lies beyond its side of the hyper-plane). C is a positive constant, called the capacity constant which indicates how critically mis-classification will be treated. This function is minimised subject to the constraints in equation 2.5. This minimisation problem is solved as a quadratic programming problem, leading to much slower model generation than with decision trees.

$$||\mathbf{w}||^2 + C\sum_{i=1}^{m}\xi_i \tag{2.4}$$

$$y_i(\mathbf{w}.\mathbf{x}_i + b) \geq 1 - \xi_i, \xi_i \geq 0 \tag{2.5}$$

Non-linear SVMs improve the potential to separate example points by mapping the input vectors, $\mathbf{x}_i$ into a higher dimensional space. The separating hyper-plane is then found in these higher dimensions. A function, called a kernel function, is applied to each example point, warping the space around the example. Perhaps the most popular kernel functions are the *Gaussian radial basis functions* (RBF) which add small peaks in the shape of a normal curve to the space for each positive example and a small trough of the same shape for each negative one. Figure 2.7 shows the application of RBFs to examples from a one dimensional space to give a two dimensional space. Two different Gaussian RBFs are shown. In (a) the very narrow RBF leads to a perfectly separable space, however, this might cause over-fitting. In (b) a wider RBF is less likely to lead to over- fitting but does not aid separability compared to the one dimensional case. The RBF is given by equation 2.6. In order to prevent over-fitting, the σ parameter is the standard deviation of the data.

$$k(\vec{x},\vec{x}') = \exp\left(-\frac{|\vec{x}-\vec{x}'|^2}{2\sigma^2}\right), \tag{2.6}$$

The runtime of training an SVM is at least $O(n^2)$ and may be higher for some parameters (Bordes et al., 2005). While SVMs may often produce better results than decision trees, the difference in runtime complexity can be substantial. In our experiments in chapter 6, we found that the SVM implementation we used could be as much as two orders of magnitude slower than the implementation of C4.5, this despite the fact that the SVM was implemented in C and the other in Java.

These last two sections have introduced the machine learning techniques applied in this thesis. The next section explains how the quality of a machine learning algorithm is judged.

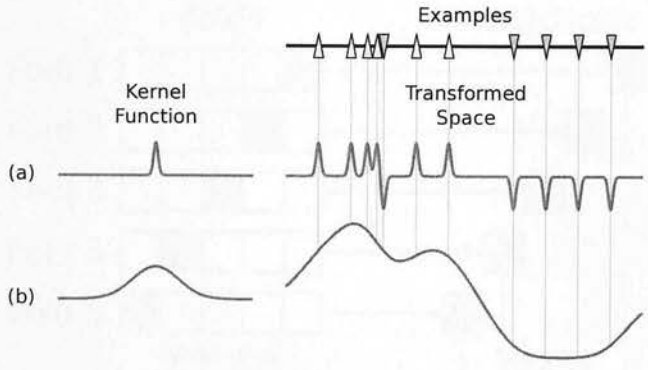


Figure 2.7: Applying different Gaussian radial basis functions (RBF) to increase the dimension of a space and improve separability. The negative and positive examples come from a one dimensional feature space (top). Two kernel functions are shown on the left and the result of applying them on the right. After applying the kernel functions, the space is now two dimensional.

2.4.3 K-Fold Cross Validation

Having chosen a machine learning technique for some problem, it is necessary to find out how well it performs. The machine learning community approves several techniques to do this, of which one is adopted in this thesis, *k-fold cross validation*. Here the example data are randomly partitioned up into k disjoint sets. A model is learned using $k-1$ of the sets as training data and then a prediction is generated from that model for the remaining set, which is termed the test set. This is repeated so that each of the sets is held out as a test set once and a prediction for the entire example suite has been created. Now, the difference between this complete prediction and the original examples can be found (either by accuracy, predicted performance or other measure). Figure 2.8 shows an example of five-fold cross validation being used to generate a complete prediction for a data set.

The most important point to note is that the machine learning algorithm is never trained over data it will be tested against. Otherwise, a simple look up table performs optimal matching. The look up table, however, fails to generalise to make predictions for new data that have not yet been seen. It is the machine learning tool's ability to generalise to new data that is of interest, not its interpolation of the example data.

There are reasons to make further refinements. If the random partitioning is pathologically bad it is possible that one test set will contain all of the examples with one classification. Since a model will be generated with no examples from that classifica-

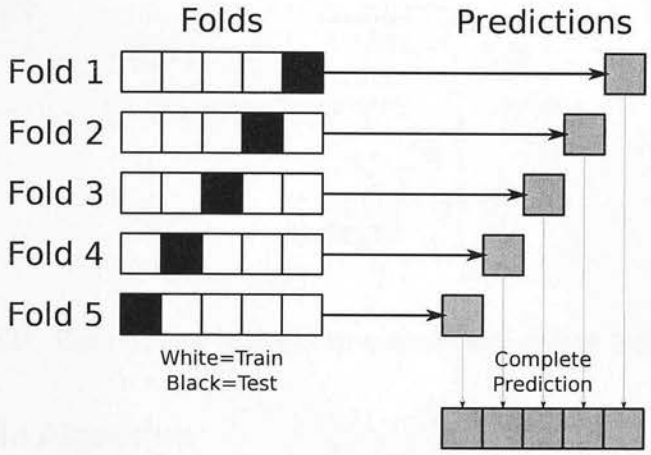


Figure 2.8: Five fold cross validation. The data set is partitioned into five sets. For each of the five folds a prediction is made for one of the sets with model trained on the other four sets. The five predictions are gathered to create a complete prediction.

tion, it will be unlikely that any machine learning algorithm will predict that classification as being possible at all; this will unfairly make the algorithm appear to perform very poorly. The partitions can be chosen to obey some desirable statistical properties, such having roughly equal numbers of each classification to improve the fairness of the partition.

This section has explained the supervised machine learning techniques and evaluation methodology used in this thesis. The next section introduces the search techniques built upon in the thesis.

2.5 Evolutionary Search Techniques

This thesis will develop techniques to search over a space a features, each of which will be expressed as a program fragment. Genetic programming and its successor, grammatical evolution, are both evolutionary search techniques that are well suited to exploring spaces of program fragments. The thesis will show a hybrid mechanism, combining the best aspects of both while avoiding their drawbacks. This section will introduce these two approaches, but first the simple, canonical genetic algorithm will be discussed, which is helpful in understanding the other two.

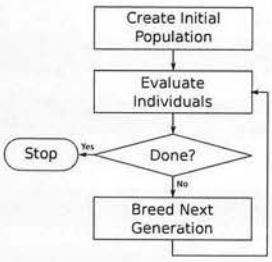


Figure 2.9: Generic flow diagram for evolutionary search techniques.

2.5.1 Genetic Algorithm

In a standard genetic algorithm (GA), potential solutions to a problem are encoded as fixed length bit strings. Each candidate solution, can be evaluated to give either an absolute fitness value which describes how well it solves the given problem or can be compared against other solutions to find which performs best. Harking back to the genetic roots of the GAs, the solution is the chromosome or genotype and its performance on the problem is the related phenotype.

To perform a GA search, an initial population of chromosomes is created in which each bit string is randomly chosen. These are then evaluated for their fitness and the fittest few selected for breeding, to create the next generation. This generational procedure continues until some stopping condition is met, normally that either an adequate solution is found, a fixed number of generations has elapsed or some number of generations have passed without improvement. The search returns the best individual seen so far. Figure 2.9 shows the generic flow diagram that applies not only to GA but to most evolutionary search techniques.

In GA, the way the digital chromosomes are bred is reminiscent of the way biological DNA combines. The chromosomes from two breeding parents are lined up next to each other and, at some random point, snipped in two; two children are created, each with parts from both parents. This operation is called crossover and is shown in figure 2.10. Another common operation used to generate new population members is to mutate or randomly flip bits from the parent genome.

Variants of GAs are suitable for different problem domains (Goldberg, 1989). Arrays of integers or real numbers have been used when that fitted the solution better; crossover and mutation operators would be altered accordingly. Variable length genomes have been constructed to allow for problems where the amount of information needed by the optimal solution is not known ahead of time. Further enhancements

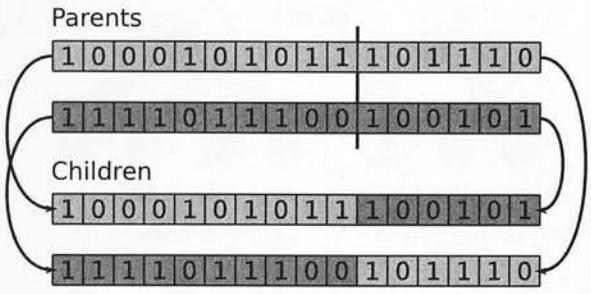


Figure 2.10: Crossover with standard genetic algorithms. Two parent genomes are cut in random places and the segments swapped to create two children.

have focussed on how the mating procedures are built; probabilities of different mating operators are often controlled by a meta evolutionary search.

2.5.2 Genetic Programming

Genetic programming (GP) is an extension of more standard genetic algorithms in which the chromosomes are no longer bit strings but expression trees which can create functions of several variables. Possible nodes that the chromosome can be made of will include the function's variables, constants and a collection of operators. Chromosome trees can be simply evaluated in a recursive manner.

GP respects the same general search paradigm as does GA (see figure 2.9) but with mutation and crossover operators that are suitable for expression trees. Crossover randomly selects a sub-tree from each parent and swaps them to create two new children, as shown in figure 2.11. Mutation involves replacing some sub-tree of the parent with a new, randomly generated sub-tree.

One of the major issues raised with GP is that because crossover will randomly select trees from parents, all sub-trees must be interchangeable. The consequence of this is that all expression nodes must return the same type, typically floating point numbers, this effect is called closure. While this causes no difficulties for simple problems, it does limit the applicability of the technique; the practitioner is sometimes forced to go to some lengths to make GP fit his problem. The closure issue is rectified in another evolutionary search technique, grammatical evolution, introduced next.

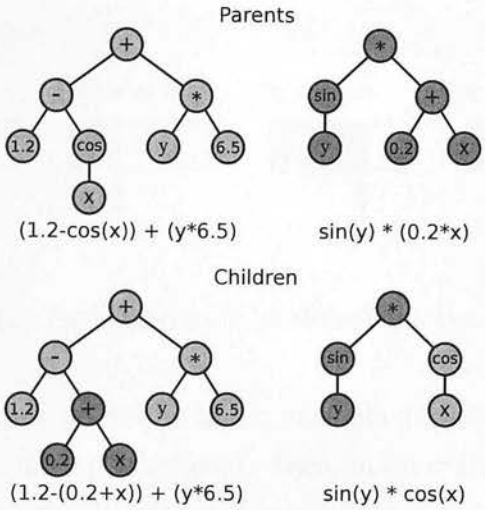


Figure 2.11: Crossover for genetic programming. Sub-trees from parent chromosomes are swapped.

2.5.3 Grammatical Evolution

Grammatical Evolution (GE) overcomes the closure limitations of GP. In GE, the goal is to find a program that solves the given problem well. The space of programs is defined by a grammar, with each program being a sentence of the language recognised by the grammar. Since the programs are then simply strings which can be from a subset of an ordinary programming language, they are usually easy to evaluate with whatever compiler or interpreter is available. For GE, with complex programming structures easily expressible with grammatical rules, there is no difficulty with types.

The chromosomes in GE are bit strings evolved by a variable length GA which simplifies the implementation as existing tools can be used to drive the search. The chromosome must then be interpreted against the grammar to construct the program that can then be run to find the fitness. The key insight of GE is that while expanding sentences from a grammar, there is a choice to be made each time a rule has more than one production; these choices can be made by reading the requisite number of bits from the chromosome. If all the bits in the genome are used up, the evaluation returns to the beginning of the bit string. This may lead to non-termination; the grammar might be on same rule that it started with. GE systems must trap non-termination by permitting only at most a fixed number of runs through the chromosome.

An example showing a program being produced is presented in figure 2.12. There is first a grammar representing expression trees of binary plus operators over symbols

```
1 <p> ::= <p> "+" <p> | "x" | "y"
```

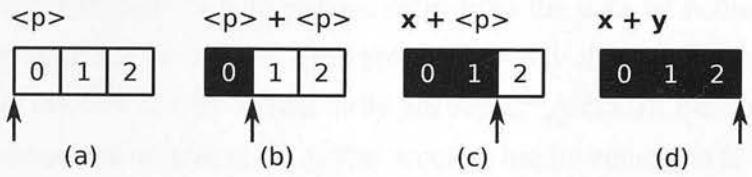


Figure 2.12: Evaluating a GE bit string against a grammar.

x and y. A binary string is given (but in this example the bits will all be grouped in pairs to select from the three productions). Then, in (a) evaluation begins with root non-terminal of the grammar and the read head at the start of bit string. In (b) the non-terminal is replaced by the zero-th production and the read head is advanced. In (c) the first non-terminal is replaced by the first production and again the read head is advanced. Finally the same process replaces the last non-terminal. No more non-terminals need to be replaced so the expansion terminates.

GE is a much more powerful system than GP but it suffers from its own drawback. While in GP small changes to the chromosome do not affect the interpretation of other parts of the chromosome, the same cannot be said of GE. Since the chromosome bit string which is used to choose amongst productions is read in one direction, a change to early bits will alter the remainder of the program. Even the smallest change can have a dramatic effect on the program. This problem is termed non-locality or position dependence by the GE community (Azad, 2003).In typical genetic algorithms, the search will begin by searching somewhat randomly and then focusses in on good areas of the space, fine tuning its solutions. GE, however, due to non-locality, can easily degenerate into nothing more than a random search, never being able to fine tune.

2.6 Statistics

This section briefly introduces some topics from statistics that are useful for this thesis.

2.6.1 Outliers

An outlier is an observation that is far from the main body of a distribution. If a small sample happens to contain an outlier then the sample mean may be thrown out from the true distribution mean, leading to erroneous conclusions about the distribution. To

be robust, a statistical measure should able to cope with outliers.

One approach to outliers is to remove them from the data set before computing the necessary statistics on the set. This presupposes that the question of which observations are outliers can be satisfactorily answered. Although there is no agreed upon, mathematical definition of an outlier, an often used formulation is based on the *interquartile range* (IQR).

Quartiles are points which separate the observations in a sample into four quarters. The first quartile, Q_1, is a point which is greater than the first 25% of the sample's observations and less than the last 75%. The median, Q_2, splits the observations at the 50% mark while the third quartile, Q_3, splits at the 75% mark. The interquartile range is the difference between the first and third quartiles, $IQR = Q_3 - Q_1$.

The standard, IQR based, outlier classifier specifies that any observation outside the range in equation 2.7 is an outlier. The range extends beyond the first and third quartiles by some multiple of the IQR. Typically $r = 3$ is considered to specify only extreme outliers and $r = 1.5$ to indicate mild outliers as well. Any outliers are removed from the sample.

$$[Q_1 - r * IQR, Q_3 + r * IQR] \tag{2.7}$$

2.6.2 Confidence Intervals

A confidence interval is used to describe an estimate of the location of the true mean of a distribution based on the set of observations in a sample. Certainly, a single observation will be insufficient to be confident that we have a good approximation for the true mean; it might be nowhere near the sample mean. As the sample size grows, containing more and more observations, we can be progressively more confident that the sample mean models the true mean of the distribution. As the sample size tends to infinity, the difference between the sample mean and the true mean, tends to zero.

Confidence intervals can be used to assess whether samples are sufficiently large. These statistical ranges show where the likely value of the true mean falls. A confidence interval always contains the sample mean and extends for some distance from it in each direction. As the number of observations in a sample increases the width of the confidence interval decreases; we become more confident that we can pin down the true mean to be closer to the sample mean.

Confidence intervals are also parametrised by a probability or confidence coefficient. A confidence interval with a confidence coefficient of 99% indicates that we

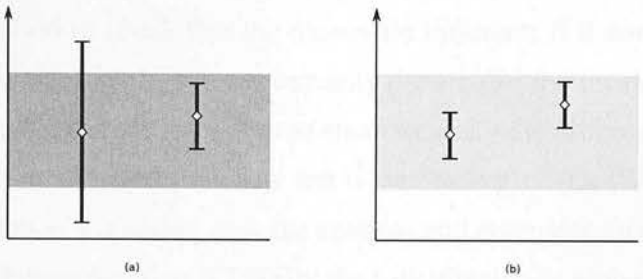


Figure 2.13: Equivalence testing with indifference regions. In (a), the confidence intervals of two samples are shown together with an indifference region. The samples are not equivalent since at least one confidence interval is outside the region. In (b), the confidence intervals are completely inside the region so the samples are considered equivalent.

are 99% sure that the true mean is inside the interval; only in 1% of trials should the true mean fall outside. Higher significance levels require wider confidence intervals; conversely, if only low confidence is demanded the interval can be very narrow.

A confidence interval is computed by the following equation:

$$CI = \mu \pm t\frac{\sigma}{\sqrt{n}} \tag{2.8}$$

Where μ is the sample mean, σ is the standard deviation of the sample and n is the number of the observations in the sample. t is the cumulative probability of the *t-distribution* for a given confidence level, α and degrees of freedom, ν - in this case $n-1$. The value t is computed by,

$$CDF_{t-distribution} = I_x\left(\frac{\nu}{2}, \frac{\nu}{2}\right) \tag{2.9}$$

with

$$x = \frac{(1-\alpha) + \sqrt{(1-\alpha)^2 + \nu}}{2\sqrt{(1-\alpha)^2 + \nu}} \tag{2.10}$$

and I_x being the regularised incomplete beta function:

$$I_x(a,b) = \sum_{j=a}^{a+b-1} \frac{(a+b-1)!}{j!(a+b-1-j)!} x^j (1-x)^{a+b-1-j} \tag{2.11}$$

2.6.3 Statistical Tests

Statistical tests are used to determine if the means of two samples are significantly different. It may be that differences observed in the sample means are due to the sample sizes being too small, rather than because the true means themselves are different. A

test can only be used to check that the means are different; if it does not declare the means to be different, that does not necessarily dictate that the means are the same, it could also be that the sample sizes are too small to verify the difference.

The most commonly used statistical test is the *Student's t-test* (W.S.Gosset, 1908). The t-test computes a 't-statistic' over the samples and compares this to a point on the cumulative distribution function (CDF) for the t-distribution (a probability distribution at the heart of the t-test). The point on the CDF is parametrised by the probability, α, and an estimate of the number of degrees of freedom in the samples. The t-test normally indicates that the means are significantly different if the t-statistic is greater than the given point on the CDF. The significance level, α, of a t-test specifies the probability that the test will assert a difference in the means when none in fact exists (called a Type-I error). The lower this value the more confident we can be that a stated difference is real. The CDF for the t-distribution is given in equation 2.9.

The student's t-test makes assumptions about the shape of the distribution and prefers it to look as normal as possible. There are also different variations on the t-test depending upon the exact use and what additional assumptions can be made. For example, if the sample sizes are equal or the variances are guaranteed to be equal, then stronger tests can be used. There are also alternatives available, for example the Mann-Whitney U test (Mann and Whitney, 1947) and Welch's t-test (B.L.Welch, 1947), which make different assumptions about the distributions under test. These alternatives, however, are generally not preferred if the student's t-test is applicable.

In this thesis, the statistical test used is Welch's t-test since it does not assume variances are equal or that the number of observations in each sample is equal. For this test, the t statistic is:

$$t = \frac{\mu_d - \mu_c}{\sqrt{s_d^2/n_d + s_c^2/n_c}} \tag{2.12}$$

and μ_i, s_i^2 and n_i are the sample mean, variance and size of the i^{th} sample, respectively.

The degrees of freedom are estimated by the Welch-Satterthwaite equation (Satterthwaite (1946)):

$$d.f. = \frac{\left(s_d^2/n_d + s_c^2/n_c\right)^2}{\frac{\left(s_d^2/n_d\right)^2}{(n_d-1)} + \frac{\left(s_c^2/n_c\right)^2}{(n_c-1)}} \tag{2.13}$$

2.6.4 Equivalence Testing

Detecting that two distributions are approximately equal is the domain of equivalence testing (Wellek (2003)). The archetypal equivalence test is based on Westlake intervals (W.J.Westlake (1972)) wherein a confidence interval is formed for the difference between two means and if that interval is completely contained within some 'indifference region' about zero then the distributions are considered equal as shown in figure 2.13. Indifference regions cannot be selected analytically, it is up to the compiler writer to express when distributions are equivalent; for example, a half percent difference in runtime may be considered negligible in some cases or significant in others.

2.7 Summary

This chapter has introduced the background techniques made use of or built upon in this thesis. It has described the typical machine learning in compilers experiment, covered machine learning and evolutionary search techniques, as well as explaining some elementary statistical methods. The next chapter presents previous work that is related to this thesis.

Chapter 3

Related Work

This chapter describes related work that is relevant to this thesis. First, section 3.1 describes automated techniques to tuning compiler optimisations. Then, section 3.2 discusses what features have been used for machine learning in compilers before covering those efforts, not in the compiler field, which have generated new features on demand. Evolutionary search techniques, which are frequently used to direct iterative compilation, are considered in section 3.3. Finally, in section 3.4, a number of attempts to bring statistical rigour to iterative compilation are considered.

3.1 Compiler Optimisation Space Exploration

Whenever compiler writers have been faced with a large number of optimisation options, they have tried to automate the proper selection of those options. This section describes the several approaches that have been applied to that problem. Section 3.1.1 covers some library specialisations which are applicable only to single domains. Section 3.1.2 details more generic attempts to search the optimisation space through iterative compilation. Then, section 3.1.3 examines techniques that made use of prior knowledge to augment iterative compilation.

3.1.1 Domain Specific Code Generation

In some cases, the author of a library may believe that the proper selection of settings for a small number of optimisations will be crucial to attaining the best performance for their code and may not trust a generic compiler to select the best settings on their behalf. Instead, the library author decides to perform the optimisations directly in their

own source code. By using source code generators, the library author can search for better quality code than the target compiler's default heuristics would achieve alone.

This technique is best exemplified by the ATLAS implementation of the BLAS linear algebra libraries (Whaley and Dongarra, 1998). The library parametrises routines for different loop tiling sizes or unrolling factors; the library's installer performs these optimisations itself, before handing the code over to the target compiler. Other routines may have several candidate implementations. During installation, the parameter space is searched to find which perform best and then source code is generated for those settings that is compiled down to the final version of the library. This unusual technique yields highly optimised code for most architectures at the expense of very long installation times because of the search.

The ATLAS approach was later modified (Yotov et al., 2003) to remove the search, replacing it with an analytical model. The model processes a machine description, containing details of architecture parameters such as cache size, and selects the corresponding inputs to the ATLAS code generator. The result is significantly faster installation time, often by as much as 30-70%. There is a cost, however, in the quality of the generated library which may not be as fast as the version that the unmodified ATLAS would have searched for; on some architectures the library ran twice as slowly while on others the difference could be negligible.

Epshteyn et al. (2005) improve on ATLAS' search by using active learning. At each stage of the search the function relating optimisation parameters to performance is estimated. The next point to sample is chosen to reflect how useful that part of the space is to the search. The point may not be the best currently predicted, but may instead be a point which improves the quality of the prediction for some part of the space which might still contain the best setting. The intelligent search technique generates similar quality code to ATLAS in about a quarter of the time.

SPIRAL (Pschel et al., 2005) is another library which tunes itself during compilation, this time focused on digital signal processing (DSP). The library is written in a DSP specific language, called SPL. The system uses search and learning techniques to select the proper optimisation strategies for the SPL code before it is converted to C code. The optimisations are similar to those used by ATLAS but are appropriate to the DSP domain. The generic specification language sets it apart from the previous approaches.

The code generation techniques here are applicable to single domains and optimise code before the generic compiler has an opportunity to optimise the libraries. The

next section describes generic techniques which change the compiler's optimisation parameters directly.

3.1.2 Iterative Compilation

Iterative compilation, in which a program is compiled with many different compilation strategies to find out which is most efficient, has been used in an ad hoc manner for decades. The technique, however, began to receive serious scientific interest in the 1990's. This section examines some of the works that have used iterative compilation.

A very limited form of iterative compilation was performed by Bernstein et al. (1989) who searched for the best of three different register allocation algorithms. The simple exhaustive search was made possible by having such a small space of choices.

Beaty (1991) searched for optimal instruction schedules with genetic algorithms. Instructions are ordered in a list, the scheduler then proceeds in a manner similar to topological sort, choosing ready instructions from the Data Dependency DAG (DDD). Whenever a choice is available, the instruction earliest in the ordered list is selected. The permuted lists are evolved by GA with the fitness function being the length, in cycles, of the schedule. The paper does not, however, report quantitative results for comparison to previous methods.

Another early paper, Walsh and Ryan (1995), described an interesting system to automatically parallelise Occam programs. Their approach uses genetic programming to search through a space of different ways to perform the parallelisation of the code to find which leads to the fastest program. However, their parallelising transforms do not guarantee correctness; instead they test each strategy comparing to the behaviour of the sequential program. This lack of provable safety limits the applicability of their system for everyday tasks. Their approach is reminiscent of the Superoptimizer, (Massalin, 1987) which exhaustively searches for small instruction sequences equivalent to a target code sequence. The Superoptimizer often produces startlingly terse code, however, due to the exhaustive search and probabilistic equivalence test, the technique is of limited practical use.

A system called GAPS (Genetic Algorithm Optimised Parallelisation) was constructed by Nisbet (1998) to search for the best loop optimisations for Fortran programs on parallel architectures. Bodin et al. (1998) iteratively searched for optimal parameters to loop optimisations (unrolling, tiling and padding) for matrix multiplication to minimise execution time. However, here the authors were not interested in the

particular search technique, seeking only to demonstrate that a search is worthwhile. They sampled only 1% of the search space yet came within 0.3% of the optimal values.

The parameters for loop tiling are searched over by Vuduc et al. (2004). They searched randomly on the optimisation parameters but offered a statistical method to determine when further observations would be unlikely to find still better results. Their technique generates a confidence value which indicates how near to the optimal value the search has managed to get; the search halts when the value is sufficiently small.

Cooper et al. (1999) use genetic algorithms to choose phase ordering sequences that most reduce code size. Their approach evolves lists of twelve optimisations chosen from a set of ten, with repeats, available in their compiler. Comparing their code size against their compiler's default settings gives very favourable results. However, the default settings are intended to optimise time, not space, and so the comparison is not entirely fair. They also manually construct a sequence of optimisations that should be space conscious and compare their technique against that. Again the comparison is favourable for sequences learned targeting the individual program. The GAs learn how to best optimise individual programs and the authors note that there is significant difference in the sequences selected for each benchmark. For each new program to be compiled the process must be repeated. The authors validate that their GA is superior to the purely random, Monte Carlo technique (MC). In this paper they found that to reach similar code sizes their GA took roughly half the number of program evaluations than did the MC version. The authors repeat a similar experiment, searching for minimum execution time (Cooper et al., 2002), in which they found once more that biased search like GAs outperformed MC search.

Phase ordering is also studied by Almagor et al. (2004). Their work aims to fully explore the space of phase orders by exhaustive enumeration. They then consider the efficacy of three search techniques over the space: genetic algorithms, hill climbers and a greedy technique. They note that they expect a predictive machine learner would make significant progress but do not attempt it. Their search finds phase sequences that create code that is 15-20% faster than the compiler's default.

Kulkarni et al. (2004) also search over phase orders but they take pains to ensure that their search is efficient. They note that many phase sequences are redundant. Some sequences contain consecutive phases in which the latter phase is prohibited by the former; the presence of the later phase will have no effect on the final code so the sequence is redundant. Other sequences can be found to generate either identical assembly code or code that only differs in the choice of registers; the latter difference

having no effect on the runtime of the compiled program. The authors discover these redundancies early and so avoid executing 84% of the runs they would need without redundancy checks.

Haneda et al. (2005) searched for compiler flags for programs. A number of different experiments are designed, each consisting of settings for each flag, either on or off, such that the set of experiments has a good mix of flags. Then after running each experiment to find the execution time of the program compiled with the given flag settings they apply a statistical test to infer which flag settings are significant. Their set up finds individual settings which are either irrelevant in isolation or should be turned on or off; interactions between flags are not modelled. Once a flag setting is determined, it is fixed for all subsequent runs, potentially removing unnecessary experiments from the table. Despite this, although they show improvement over the default compiler settings, the number of runs they perform is large.

Later, Pan and Eigenmann (2006) improved on the search time of the previous technique by starting from the case where all flags are turned on and by iteratively eliminating the flags with the greatest negative effect on the performance of the compiled program. The demonstrated that of several flag searching techniques theirs achieved a similar 7% runtime speed up for a much lower number of search points. Agakov et al. (2006) found that a simple random search could find better compiler flags than were found for the same number of evaluations than GAs.

Chen et al. (2010) consider whether iterative compilation remains effective when a program's input data is varied. They apply 1000 different data sets to 32 programs compiled with 300 sets of compiler options. They discovered that each program had at least one optimisation setting which achieved on average 86% of the maximum performance across all data sets. This indicates that, for the benchmarks in their suite, iterative compilation is insensitive to the input data and thus a valid technique.

The works described in this section all search the optimisation space afresh each time they are presented with a new program to compile. In the next section, iterative compilation is augmented with knowledge from previous runs to improve the search.

3.1.3 Iterative Compilation with Prior Knowledge

The search of iterative compilations can be biased with knowledge from prior searches to speed up the search. This section describes techniques that have worked along these lines.

Cooper et al. (2005) noticed that if they could predict the effect of an optimisation on the performance of the program they would be able to avoid executing it during the iterative search. With a model of the architecture and a profile run of the program being compiled, they estimate how the runtime of the program will change as instructions in each basic block are inserted or removed. Their approach is the ACME environment which can reduce the number of program versions that must be executed by an order of magnitude with the same accuracy of result. However, since they require the overall structure of the program to remain constant, they can only cope with a few data-flow optimisations. Moreover, their model of the architecture is quite simple, being unable to account for such things as branch prediction and cache misses.

Cavazos et al. (2006a) also predict the effect of optimisations to speed up search. In this work, however, the authors train a model offline using a neural network and a suite of training benchmarks. The model makes performance predictions based on characteristics of the new program to compile; these characteristics are formed by comparing how the program reacts to a number of predetermined transformation sequences. The authors claim that by testing the reaction of a program to only four different transformation sequences, they can estimate how that program will behave under different sequences with an error of only 7%. Later, Dubach et al. (2007) extended this idea building empirical performance models based on static code features or descriptions of the source code. They achieved a high correlation to the real performance function.

The shape of the search space is modelled in Agakov et al. (2006) to focus iterative compilation directly onto those parts of the space that are likely to hold the best performing point. The authors first train a model on optimisation sequences over a number of DSP kernels using source code features. This then predicts which parts of the space lie close to the optimum and search within that predicted subspace with Monte Carlo and genetic algorithm techniques. They showed that this way of biasing the search was able to outperform the unbiased search, needing to examine an order of magnitude fewer points.

The techniques presented in this section reduced the size of the search space by using prior knowledge of previous programs' behaviours. The resulting models where used to limit the amount of executions an iterative compiler needed to find good optimisation settings for a new program, but the search is still required. The techniques in the next section, however, use prior knowledge to directly learn the right optimisation settings so that a new program will be optimised without recourse to any iterative compilation search.

3.1.4 Predicting Optimisation Settings

While iterative compilation has shown enormous benefits, optimising programs far better than the level available through the compiler's ordinary heuristics, the process is inordinately expensive, often requiring many runs to find a good optimisation setting. Researchers have, instead, sought to replace the hand tuned heuristics with ones that are automatically created from empirical data. Using the results of many prior trials of different benchmarks with different optimisation parameters allows the compiler writer to create heuristics that achieve much of the benefit of iterative compilation without the cost. There is a cost, however, which is that the iterative compilation is done a priori, at the factory, to construct the automated heuristics. This section describes various approaches the have been tried in the past.

Balasundaram et al. (1991) used a simple model to estimate the benefit of various data partitioning schemes for parallel programs. The model was hand built with only the thresholds and constants learned from iterative compilation experiments, but it was certainly one of the precursors of modern machine learning in compilers. Calder et al. (1996) used neural networks and decision trees to learn branch prediction heuristics; these heuristics were to be used in both hardware implementations and compiler optimisations. Branch prediction has the advantage of being unaffected by sampling processes and is completely noiseless. However, this early technique performed well, yielding a mis-prediction rate of 20% compared to the best alternative at the time which only managed 25%.

Paterson et al. (1997) applied genetic programming to developing caching algorithms but found that the training data was over-fitted, leaving poor performance on the test set. O'Neill and Ryan (1999) employed grammatical evolution to a slightly more complex version of the same problem and found no over-fitting. Indeed, the learned caching algorithm suffered fewer than 25% the number of cache misses of the standard LRU cache replacement policy.

Stephenson et al. (2003b) present a system for genetic programming on the hyperblock formation problem and on register allocation. This was later extended in Stephenson et al. (2003a) where they demonstrated FINCH, their generic compiler software that could replace priority functions with ones found by genetic programming. In their system a compiler writer would present features for the desired heuristic and consume the result. The GP driver then searches for a good heuristic. They show their system achieves some impressive speed-ups on a variety of case studies. This was

one of the first works that attempted to generalise across different optimisations.

Moss et al. (1998) was another early work to use machine learning to replace compiler heuristics. They were concerned with instruction scheduling and created a model which would predict, given a sequence of already scheduled instructions and two candidates to be placed next, which of the two would be more profitable. The authors used a number of supervised machine learning techniques, including decision trees to build their models. Mcgovern and Moss (1998) noted that the previous technique required an optimal and expensive scheduler to generate good training data. Instead they demonstrated that unsupervised learning could be used as well with a reinforcement learning technique.

Monsifrot et al. (2002) used decision trees to learn when to unroll loops. They learned only a binary decision, leaving the choice of unroll factor to the underlying compiler. They partitioned loops from the training set into those which would benefit from unrolling, those which would not and ignored those for which the difference was insufficiently large. They were able to outperform the default, hand-tuned compiler heuristics on two different platforms, demonstrating the potential of machine learning for compilers.

Cavazos and OBoyle (2005) learned improved values for five integer constants inside of two of JikesRVM's inlining predicates. The inlining predicates are composed of a number of thresholds over characteristics of a method's byte-code. The characteristics are the callee size, inline depth and caller size. Tests against these thresholds lead to a fixed set of *if-then* rules determining the inlining policy of the JVM. The thresholds are learned via a genetic algorithm to minimise running time, compilation time or a balance of both, with performance being taken as the geometric mean over the their chosen benchmarks. The authors generate impressive speed-ups on their chosen benchmarks against the default constants in the JikesRVM. They also apply their technique to different architectures, demonstrating that they can learn appropriate values in different environments.

Turning to a different problem in the same JIT compiler, Cavazos and Moss (2004) decide whether or not to schedule instructions in basic blocks. Instruction scheduling is an expensive operation that must be paid for at runtime in JITs. Thus, the cost of the optimisation must be compared to the potential gain in performance of the optimised code. The authors use a supervised learning technique called rule set induction to generate a predictive model that can be executed quickly. Good results are reported compared to the standard heuristic. This is, perhaps, surprising since the benefit of

this optimisation is proportional not only to how much faster the basic block can be made, but also to the number of times that it will be executed during the program; the cost, however, is not proportional to the execution count. The learning algorithm, on the other hand, has no idea how often each basic block will be run nor whether the program will even ever terminate, yet the authors report excellent numbers. The same authors (Cavazos and O'Boyle, 2006) later applied logistic regression, a form of supervised learning, to choose what level of optimisation was worthwhile for each method in the virtual machine. Again, they get impressive results of around 25% reduction in execution time, but no discussion is offered as to how the total running time of the program affects the compile time, execution time ratio.

Cavazos et al. (2006b) use rule induction to choose between register allocation algorithms, Linear Scan (LS) and Graph Colouring (GC), in the Jikes RVM. In the context of a Just-In-Time compiler expensive optimisations, GC for example, must be applied only where the benefit justifies the cost. The authors thus attempt to learn the appropriate cost-benefit relation. The authors select a number summaries over the control flow graph of each method as inputs to their learning algorithm. Rule induction automatically ignores irrelevant features. Methods are labelled according to which optimisation is better. Methods are labelled GC if the spill count is greater that that for LS by some threshold or if the cost is only worse than for LS by some threshold. This labelling is what is to be learned. The authors present one of the rules learned for particular threshold settings. Of 142 methods that should have been categorised as GC, only 64 were, 45%. However, of those 1844 that should have been LS, 1815 were so chosen, 98%. No analysis is offered for the poor categorisation of GC methods.

Stephenson and Amarasinghe (2005) employ two different machine learning techniques to loop unrolling. They regard the loop unrolling problem as one of multi-class classification by limiting unroll factors to fall between one and eight inclusive. So phrased, they use a Nearest Neighbour classifier and a Support Vector Machine to distinguish amongst cases. They learn the optimal factor 65% of the time and the first or second best for 79% of the time. They get 5% improvement in runtime for SPEC 2000. These impressive results, they say, were achieved in a few seconds of compute time, compared to many years of human tuning in the original compiler. Despite also spending two weeks instrumenting the compiler and one week collecting data the total time is far better than in hand tuned cases.

Support Vector Machines are used by Tournavitis et al. (2009) to suggest OpenMP parallelisation strategies to the user. Unlike traditional, conservative automatically

parallelising compilers they do not guarantee any correctness that the code can, in fact be parallelised. This optimistic approach is justified since after close inspection by a human, all loops in their test benchmarks were parallelisable.

Long et al. (2004) use instance based learning to choose transformations from Pugh's unified transformation framework on Java programs. They are able to achieve 70% of the performance available through an exhaustive search.

Thomson et al. (2010) reduce the number of programs that must be executed to gather training data by first clustering benchmarks according to their features and then building their models using only the centroid of each cluster. They achieve a speed up of 1.14 for EEMBC benchmarks after training on only six programs.

Dubach et al. (2008, 2009) learn good optimisations settings across a family of configurable processors. Their motivation is that knowing the performance of benchmarks on as yet unrealised architectures is crucial when exploring the space of different processor designs. However, once a processor is selected the next step would be to tune a compiler for it which may then change the cost benefit analysis for the architecture. The authors, instead, learn how to predict how an optimising compiler would operate for a new architecture, thus speeding up the search of architectures.

Maoa and Shen (2009) learn, for individual applications how different input sets will affect the program. They use their predictor to select different optimisations, suitable for each given input, in JikesRVM. Since input data can take any, arbitrary form, the authors provide a language which permits the application developer to specify the important features of the data.

Finally, Lokuciejewski et al. (2010) use evolutionary search to find the best machine learning algorithm to use. Their space of algorithms included decision trees, support vector machines and nearest neighbour methods, together with their various parameters. They search over this space to find the algorithm which generates a model with the highest accuracy when learning whether to apply loop invariant code motion optimisations. Their system selected a machine learning algorithm with 67% accuracy, compared to 54% for the worst algorithm's accuracy.

This section has introduced several of the applications of machine learning to replacing compiler heuristics. The next will cover the ways in which features have been used by machine learning in compilers works before going on to describe works outside of the compiler field which are able to generate features on demand.

3.2 Features

This section discusses the types of features that have been used by prior works for machine learning in compilers. The latter half of the section discusses the measures that have been taken to be selective about features and approaches that generate new features on demand.

3.2.1 Static Code Features

The majority of machine learning in compilers work has used features which summarise the source code without running the program first. In all of the examples below the features are fixed and applied only to individual problems.

The features of Calder et al. (1996), when learning branch prediction routines, include information about both the context of the branch and the two successors of the branch. Their branch predictor can be used in both hardware and compiler optimisations. They have features describing the types of the branch and successor instructions, whether the branch is a loop branch, its direction and so on.

Stephenson et al. (2003b), concerned with hyper-block formation, use a variety of features, including the maximum dependency height of instructions in the path, total number of instructions and whether there are memory hazards. For register allocation, they use the number of calls in the containing basic block, use-def counts and estimates of spill costs and benefits.

Cavazos and Moss (2004), while learning whether to inline methods in a JIT, use instruction type histograms as the majority of their features. They also use data available to the JIT, including how many of a method's instructions are garbage collection points or can cause a thread switch.

When predicting whether to unroll loops, Monsifrot et al. (2002) summarise loops according to the number of memory accesses, histograms of the different instruction types and iteration count estimates. Stephenson and Amarasinghe (2005) also learn a loop unrolling heuristic. Their features are a super set of those for Monsifrot et al. (2002) with information about use-def chains and dependencies added. Their full list of features can be found in table 6.10. For instruction scheduling, Moss et al. (1998) used features describing the types of instructions, dependency height and instruction latencies.

In these works, the compiler writer has manually constructed features that they think may be important when deciding the heuristic or that are easy to compute in their

compiler of choice. Rarely is there any justification for the presence of their features beyond an appeal to intuition. The next section describes a few works which have built generic machine learning compilers and so have had to leave the choice of features open ended.

3.2.2 Generic Feature Systems

Some works have focused less on fixing feature sets and more on providing simple mechanisms to allow the compiler writer to indicate their own static features.

Stephenson et al. (2003a), in their FINCH software, allow any priority function to be replaced with a genetically programmed version. The compiler writer tells FINCH what priority function to replace and lists the features they wish to use as arguments to the function. Only scalar arguments are permitted so the compiler writer must do all of the work of summarising the program before handing off to FINCH.

The approach taken by Fursin et al. (2008a), who use machine learning to predict the compiler flags, take pains to ensure that specifying features is easy. They extract details of the compiler's internal representation which they store in a relational database. Once the data are available, they process it with a Prolog like language to produce the features. The simplicity of the language allows new features to be added to the system easily. The current incarnation of their tool offers features that summarise the counts of different types of instructions, aspects of the control flow graph, constants and memory references; third party developers are encouraged to submit additional features and have been doing so.

Maoa and Shen (2009) present a feature specification language, XICL, which allows application developers to compute features over arbitrary data sets. Their system attempts to make common scenarios easy to specify and allows more complex cases to be handled by the full computing abilities of Java. Their system does not combine features in any way and the developer is completely responsible for creating features over structured data.

While these works have not fixed their feature sets for engineering reasons, they require the compiler writer to follow that course at some point. The systems presented in this section were more generic versions of the special cases from the previous section. Next are presented a few works that have gone beyond static code features and have looked at features that can only be gathered after a profiling run.

3.2.3 Dynamic Features

This section describes feature approaches which use information only available after running a program in one or more profiling runs.

Cavazos et al. (2007) used the values of performance counters as features. The counters describe such things as the total cycles, cache misses, branch mis-predictions and so on. They use a simple logistic regression as their machine learning algorithm. They test their technique on SPEC 2000 and MiBench benchmark suites and find that they compare favourably to the work of Pan and Eigenmann (2006), who used static features for the same problem.

Triantafyllis et al. (2003) add iterative optimisation search to the the Intel Itanium compiler. They collect good compiler optimisations by considering that how a program reacts for one optimisation will be similar to how it react to a similar optimisation. Their technique permits them to rapidly search the space since they do not need to execute the programs, their estimator takes that role.

Cavazos et al. (2006a) dispense with static code features altogether. Instead they compile and run a program with a few canonical optimisation sequences and create a vector describing the execution time *responses*. These response vectors are then the features used to learn performance from. The authors show that their technique outperforms a static code feature method but it is not clear if this is because the static features are poorly chosen. On the other hand their reaction based approach is simple to implement and is both compiler and architecture agnostic, something which cannot be said of most other feature systems.

These papers used dynamic information about a program to construct machine learning features. The next section examines how, given a set of features of which not all may be good, the compiler writer can remove those poorly performing features.

3.2.4 Feature Selection

In a typical experiment, the compiler writer will implement a number of features he believes will be pertinent to the heuristic he wishes to improve. Often there will be a large number of features that are based on either hunches, intuition or ease of implementation. The interaction between features and a machine learning algorithm is complex. Features that are based on human intuition may not be the best features to choose. Features may not represent all of the relationship between the program and the desired outcome or, even if they do, they may not work sufficiently well with the

machine learning algorithm.

Features Selection assumes that the problem description language contains a superset of the features needed to pick out the target hypothesis, i.e. that the researcher starts out with more features than are required and that some must be removed or merged. The simplest approach is to enumerate all sets of features and determine which yields the best machine learning performance. Whilst always providing optimal results, this exhaustive search is exponential in the number of features and hence, typically, impractical for all but the most trivial problems.

Heuristics are therefore employed. Common hill-climbing searches are Forward Selection (FS) and Backward Elimination (BE). The former greedily adds features while the latter removes them. Genetic Algorithms (GA) are often used when local minima trap the simpler greedy algorithms. Kohavi and John (1997) present a survey of feature selection techniques. They also demonstrate that the true quality of the features can only be found by directly asking the machine learning algorithm to make predictions using the features and seeing how good the predictions are. They define *wrapper* methods to be those which use the quality of the machine learned model to drive the selection process, while those which do not are termed *filter* methods.

Vafaie and DeJong (1993) consider how to choose features for learning image classifications. The authors search for binary strings representing which features to provide to classification algorithm from a choice of 100. To evaluate these length 100 binary strings the classification algorithm, AQ15, is trained over the selected features and then tested against a data set to determine its recognition performance. Two feature set are compared by the recognition performance they yield. The paper compares GA and sensitivity analysis (Sequential Backward Selection, SBS, which greedily removes features while doing so improves fitness). GA performed well when there are interactions between features and local minima but was inefficient when there were few interactions and local minima. In all cases, however, GA produced excellent quality results.

In the field of compilers, scant attention is given to feature selection. Dubach et al. (2007) first removed features for which there was no variation and then used principal component analysis to further reduce the number of dimensions in their features. Cavazos et al. (2007) use *mutual information* to analyse the contribution from each of their features. Monsifrot et al. (2002) used a wrapper technique, training a neural network on progressively fewer features, stopping when the prediction accuracy began to drop. Stephenson and Amarasinghe (2005) employed both a mutual information technique and a greedy, forward selection algorithm.

While feature selection allows redundant and spurious features to be removed, speeding up learning and often improving accuracy, it cannot assist in those cases where there is information missing from the initial set of features. The next section describes techniques that can adaptively add features.

3.2.5 Feature Generation

When the compiler writer implements a set of features, he must make sure to include all relevant information. No machine learning algorithm will construct a good predictive model if the features it is given to work with do not correlate with the target heuristic. While feature selection techniques will mitigate the presence of unnecessary features, missing information is much harder to fix. Despite the complexity of the compiler's internal representations and the clearly infinite number of potential features, there has been *no* work on generating features in this field.

Constructive Induction (CI) pairs a feature management system with a selective learning system. If it detects that the learning system's performance is too low, it determines that composite features are needed. Good composites are then searched for. Standard CI does not consider removing features. Bensusan and Kuscu (1996) use Genetic Programming (GP) to learn two features to add in their CI system. Their example is learning the four-bit parity function (notoriously hard for selective learning systems). They allow their GP system one type of function node, XOR, and search for expressions that allow a feed forward neural network to learn the parity function. The authors compare their CI system performance only with selective learners. The CI system is 100% accurate while the selective learners are 0% accurate. It should be noted, however, that GP is perfectly capable of learning 4-bit parity all by itself, particularly when provided only XOR. While the authors' approach is certainly interesting, it would surely benefit from being applied to a real world example.

Ritthoff et al. (2002) present a combined feature selector and generator. Their starting point is to have some problem to learn with a large number of features and a set of generators which can combine those features to create new, composite features. They also have some machine learning algorithm (in the paper they use a Support Vector Machine (SVN) but any will do) and example data. Their approach is to build a modified variable length Genetic Algorithm. Individuals contain a list of n strings describing features. Each of these is either an initial feature or a composite of features created by some applications of the generators. Additionally, an n bit vector describes which

features will are selected and so will be provided to the SVN to learn over. The standard genetic mutator (bit flipping in the selection vector) and one-point variable length crossover are augmented with a feature generator mutator. This takes an individual genome and adds some number of features generated from existing, selected features. For example, it might choose features x and y from an individual and add feature x * y to it. Many features may be added in one step and they may combine newly added features as well to permit arbitrarily complex features to be constructed. Only selected features may be combined in this way. Evaluation of individuals tests the classification performance of an SVN trained over the example set with the given selected features. Their results on artificial problems show that their combined approach clearly outperforms learning over insufficient feature sets. That is not, perhaps surprising. They also show learning over a large time series (5000 points) for estimating coefficients in chromatography experiments. They show that their hybrid approach is very successful at helping the SVN to learn these coefficients for different time series. Design of the hypothesis language and generators will still influence the performance of the system. However, this approach allows the easy incorporation of domain knowledge since hand crafted features and generators are simple to add.

Mierswa (2004) uses Genetic Programming (GP) to learn features over time series data, particularly audio sequences. Three genres of operation are provided to the GP system. These are:

- Transformations - which map each value in the input stream to a new value.
- Functions - which aggregate a stream into a fixed length vector.
- Windowing - which applies functions to moving windows over a stream to create a new stream, with fewer elements.

Expressions of these functions are combined to create features to be used in the C4.5 Decision Tree algorithm and an SVN. Experiments classify audio streams according to user preference or genre (for example, pop versus techno). The features learned are not obvious but it is not clear how successful these are compared to human feature generation attempts. Consequently, while the paper demonstrates that features may be learned over infinite time series there is no clear idea how well it performs.

This section has shown that the choice of features to use in compiler experiments has been somewhat ad hoc to date. Although selection has been used to remove features, no attempts have been made to create features when needed. In other fields,

feature generation has most focused on time series, not structured trees and graphs as are present in compilers.

3.3 Evolutionary Search

Iterative compilation requires searching a large parameter space to find the best settings for compiler optimisations. The space is, on occasion, so large that exhaustive enumeration is impractical. Researchers need search techniques in those instances and as shown in the last sections, genetic and other evolutionary search algorithms are often the approaches used. This section describes some of the most important evolutionary search techniques for this thesis.

3.3.1 Genetic Algorithms

Perhaps the first use of biologically inspired genetic algorithms dates back to Barricelli (1957), who was interested more in experimenting with artificial life than searching. The techniques did not become widespread for some time, until the book of Holland (1975). Holland introduced a *schema* theorem which showed how the crossover of binary strings could lead to some portions of the search space being unreachable if sufficient random information is not present at the beginning of the search.

3.3.2 Genetic Programming

In Koza (1990b), John Koza presents the original Genetic Programming (GP) system. Breaking from the typical linear, fixed-length Genetic Algorithms (GA) he searches for Lisp-based S-expressions to solve problems. The essential genetic operators, mutation and crossover, are introduced over trees. A typical mutation selects some sub-tree, removes it and replaces it with a new, random tree. Simple crossover randomly chooses two sub-trees in the parents and swaps them.

Koza's GP places severe restrictions on the form of the expression nodes that are searched over. In particular, since crossover implies that two arbitrary nodes from different chromosomes will be swapped, the return type of the expression that each node represents must be the same, for instance all expressions might compute real numbers. This limits the expressive power of GP. Constructing subroutines is another problem for GP, and Koza's solution is to predetermine the maximum number of subroutines available to the programs, which again limits the power of GP.

To combat these issues, several alternatives have been created. One interesting approach is from Spector et al. (2001). Here, rather than searching over Lisp style expressions, the programs are instruction streams for a stack machine. Spector avoids the single type issue by allowing multiple stacks, one for each type. In another advance, the evolutionary programs here create their own children, so that the parameters of mutation, and crossover need not be specified by the researcher.

3.3.3 Grammatical Evolution

Ryan et al. (1998) introduce a novel version of evolutionary search, called Grammatical Evolution (GE). In typical GP the genotype and phenotype are identical, usually lisp-like expressions. By contrast, in GE the genotype is a list of choices to make in the expansion of sentence from a Backus Naur Form grammar (BNF).

Separating genotype from phenotype allows GE to use any search technique suitable for variable length integer lists, widening the appropriate choices considerably over GP. Additionally, due to the phenotype being any grammatically correct string of symbols, the system can easily produce program fragments in any language, making fitness testing trivial by comparison to the frameworks needed for non-lisp GP.

However, the genotype in GE is a linearisation of grammatical expansion choices. As such, it suffers from a number of setbacks. There is a ripple effect due to the fact that position in the genome is not understood by cross-over or mutation. This means that small changes to the front of a genotype can easily cause massive changes to the interpretation of the remainder. It is difficult to explore neighbourhoods of individuals. Additionally, when more genetic material is required during the expansion of a rule than is available in the individual, the genotype is wrapped. This often causes infinite expansions leading to many individuals that cannot be mapped to phenotypes(Ryan et al., 2002). Finally, equal weighting to production probabilities forces the system to be highly sensitive to the grammar used. Two different grammars, both recognising the same language can radically alter the probabilities of various strings being produced. The user has little recourse in this system to overcome this drawback.

Unfortunately, while the authors demonstrate their system over standard symbolic regression problems, they do not present quantitative results, making this approach difficult to compare to ordinary GP.

In an attempt to solve some of the problems inherent in Grammatical Evolution (GE), Ryan et al. (2002) propose a new formulation, the Chorus System. Chorus dif-

fers from GE in the mapping of genotypes to phenotypes. In GE each choice in the genome is decoded modulo the current number of choices in the grammar expansion. In Chorus, on the other hand, all genes are modulo the total number of productions. An initially empty concentration table is maintained. Each time a choice is encountered in the grammar expansion, the table is consulted and the production with the highest positive concentration is selected and its concentration reduced. If no such production exists, the genome is read, each gene incrementing the concentration table until a suitable production can be found. Wrapping is not used, individuals with insufficient genetic material are heavily penalised.

While the new approach introduces a much higher percentage of introns than in GE, it ensures that the absolute position of genes is irrelevant. In the paper, the authors develop a complete schema notation based on regular expressions for their system. Results are shown comparing against the standard GP benchmarks. Chorus is out performed by GE and occasionally by GP. The authors, however, expect that the high degree of introns contributes to this poor performance, yet expect that to be a benefit in longer trials.

This section has introduced works on various evolutionary search techniques.

3.4 Statistical Sampling and Sequential Analysis

This section discusses attempts that have been made to bring statistical rigour to iterative compilation and machine learning for compilers.

3.4.1 Statistical Rigour in Execution Time Measurement

Iterative compilation is expensive when use to tune the compilation of a single program. Its use in machine learning is to generate as much training data as possible and lots of programs have to be iteratively compiled. Researchers may need months of compute time to complete their experiments. This problem is compounded since noise in execution time measurements mean that multiple runs of each program version must be performed. To the best of our knowledge, all prior machine learning in compilers use only fixed sized sampling plans where a constant number of runs are executed.

Efforts to promote statistical rigour in execution time measurements have been made(Georges et al., 2007; Blackburn et al., 2006). In these, a program version is run multiple times until either an estimate of inaccuracy is sufficiently small or some max-

imum number is reached. Each point in the optimization space is executed until a good estimate of its mean so the data is statistically valid. Specifically, the stopping criterion is that either some maximum number of runs has been reached or that a confidence interval of the sample is less than some fraction of the mean. However, this effort does not take into account the relative merits of each point. A point that is clearly bad will be refined just as much as the most promising point in the space. Since their technique considers each point in isolation it can perform worse than an optimally chosen constant sized approach.

In (Mytkowicz et al., 2009), the difficulties of avoiding measurement bias are described. The authors demonstrate that, even with a simulator, apparently innocuous modifications (such as sizes of irrelevant environment variables) can affect the performance of a program. They suggest that random changes must be made to the set up state so that multiple measurements are required. Even in simulators, previously thought to be a source of noise free data, correct measurements must handle noise.

3.4.2 Sequential Analysis

Sequential analysis, however, has been used to reduce the cost of sampling in contexts from industrial processes(Wald, 1947) to medical trials(Whitehead, 1992). In (Maron and Moore, 1994, 1997) wherein machine learning models are 'raced' to find which one is best by using leave one out cross validation; an initial set of candidates contains all the models, then in each step models that perform poorly are removed and if one remains it is declared the winner, otherwise the candidates are tested on another point from the leave one out cross validation. Their work, however, relying on Hoeffding's inequality(Hoeffding, 1963), requires that the random variables under consideration are all bounded - which is not the case for run times. Moreover, their work only concentrates on removing poor performers, it does not consider the situation where some of the random variables are equivalent for practical purposes.

3.5 Summary

This chapter has presented prior work related to machine learning in compilers. The majority of works have followed a similar pattern; a few static code features have been chosen based on intuition, some example programs are compiled with different heuristic values and run a constant number of times - a machine learning algorithm

then creates a predictive model to replace the heuristic.

Few works offer generic compiler support for machine learning. Those that do often restrict themselves to just part of the problem. Chapter 4 will demonstrate a compiler system for GCC which provides all of the necessary capabilities to handle machine learning in compilers experiments.

Occasionally, works have used feature selection to solve the problem of having too many redundant features. No work has ever sought to address the converse problem in which the features do not contain sufficient information. All machine learning in compilers works have been limited by the amount of correlation the features can contain. Chapters 5 and 6 will show how new features can be generated on demand.

Machine learning in compilers experiments require large amounts of iterative compilation to source their training data. The requirement to get statistically sound data is either ignored or handled by running program versions an unnecessarily large number of times. In chapter 7 is an algorithm which adaptively manages the number of executions required for iterative compilation to ensure that statistically sound data is acquired with the minimum number of runs.

Chapter 4

Fine Grained Extensible Compiler

This chapter describes the development of an extensible, machine learning enabled compiler, based on GCC. No current production compiler is sufficiently powerful to support machine learning experiments, so this new compiler was created; it was used for all of the experiments in this thesis. More details can be found in appendix A.

4.1 Introduction

Today's compilers do not provide the functionality required by machine learning techniques. All the main compilers were designed before machine learning was shown to be useful to their goals and this has left a set of engineering problems that must be overcome by any researcher who wants to explore the potential of machine learning in compilers. The researcher needs to affect the compiler across a wide range of granularities; sometimes forcing it compile code differently and sometimes extracting information about the program as it is compiled.

For control purposes, compilers mostly go no further than providing a range of global settings, typically through command line options or environment variables. These, however, do not allow the researcher to change the compiler's response to individual functions, loops or other constructs. Unless the current experiment is at the whole program level (or at least at the compilation unit level) then these settings will not be sufficiently fine grained. Occasionally, more capabilities are made available through source code annotations and extensions such as pragmas in C. However, being forced to modify the source code for large benchmarks is likely to be error prone and difficult to automate. Instead, researchers make ad hoc changes to the compiler, typically on a per experiment basis, with the result being difficult to maintain and share

with others.

What is needed, instead, is a compiler that allows itself to be easily extended; so that its heuristics may be replaced and information recorded about each compilation both in the fine and coarse grained behaviour of the tool. This should be possible without the researcher needing to alter a single line of source code and should observe best practice in software engineering. This chapter presents just such a compiler, modifying GCC to allow complete control over the aspects of the compiler which are necessary for machine learning experiments. The compiler provides extension points that expose the behaviour of heuristics and enables those heuristics to be overridden without the need for source code changes to the underlying compiler. From a software engineering point of view, the compiler becomes a modular collection of interoperating, encapsulated components to which new components can be added without altering the existing set of components. GCC becomes a fully capable research compiler.

The remainder of this chapter is organised as follows. Section 4.2 shows, as the situation stood before the work of this chapter; how the alterations necessary to achieve a typical machine learning experiment lead to unmaintainable code. Then, section 4.3 introduces the solution, *libPlugin*, which makes GCC extensible. Finally, section 4.4 revisits the example of section 4.2, showing how *libPlugin* enables the experiment to be completed in a modular, maintainable manner without modifying the compiler .

4.2 Motivating Example

In this section we show how a current compiler, not constructed with extensibility in mind makes machine learning experiments difficult to implement. In particular, the compiler's own source code must be altered to support the changes required. The result will be that the compiler's once clean source is obfuscated by code for just this experiment; the situation will be error prone and difficult to maintain.

The section will consider an example experiment, learning proper loop unroll factors, and will examine the changes the compiler writer must effect. The example experiment is broken down into phases and each is described in its own subsection. First, section 4.2.1 describes the original heuristic that the experimenter wishes to replace. The set of loops to unroll for iterative compilation will be considered in section 4.2.2 with the iterative compilation itself covered in section 4.2.3. Static code features for each loop will be computed in section 4.2.4 and then a model will be learned and used to replace the original heuristic in section 4.2.5.

```
1 int decideUnrollTimes(loop* lp) {
2     int times = /*the heuristic*/;
3     return times;
4 }
```

Figure 4.1: Original unrolling heuristic before the machine learning in compilers experiment.

4.2.1 Original Heuristic

The compiler writer has decided to replace the loop unrolling heuristic of GCC. The heuristic is somewhat spread across the several functions, however, its essence is depicted in figure 4.1. The original heuristic returns the number of times each loop should be unrolled or zero if the loop should not be unrolled. A small amount of refactoring allows it to look like the pseudo-code presented here and in GCC, the real heuristic returns not only the number of times to unroll a loop, but also which of several different flavours of unrolling to perform. However, despite this, the unmodified heuristic is clean and comprehensible.

Now that the researcher has identified the heuristic to change and being without an extensible compiler, he must hereafter effect his modifications for his experiment directly in the compiler's source code, obscuring the original heuristic, as will be shown in the following sections.

4.2.2 Determining Unrollable Loops

The first phases of a machine learning in compilers experiment use iterative compilation to gather the example data showing the impact of different heuristic choices. The data will be used to learn a model over and the model will, in turn, replace the heuristic. The search space for the iterative compilation must be determined.

For the loop unrolling example, the researcher decides that at each point in the iterative compilation only one loop per function will be unrolled, the rest will remain unchanged. In this way, he hopes to reduce the interactions between loops which might affect data gathering and to make counting the number of cycles spent in the changed loop easier to measure. To gather the example data, he must compile the programs many times with different unroll factors for the loops and to do that he must know what loops each program has and which are unrollable. Thus, the compiler writer adds code to the heuristic to list the unrollable loops. The changes are similar to that in

```
int decideUnrollTimes(loop* lp) {
    int times = /*the heuristic*/;
    if(shouldPrintLoops) {
        print("unrollable-loop=%s,fn=%s\n",lp,lp->fun);
    }
    return times;
}
```

Figure 4.2: Unrolling heuristic augmented to print unrollable loops.

```
int decideUnrollTimes(loop* lp) {
    int times;
    if(shouldOverride) times = /*search in override file*/;
    else times = /*the heuristic*/;
    if(shouldPrintLoops) {
        print("unrollable-loop=%s,fn=%s\n",lp,lp->fun);
    }
    return times;
}
```

Figure 4.3: Unrolling heuristic with the ability to override the settings added.

figure 4.2.

4.2.3 Iterative compilation

From the list of unrollable loops, the researcher can build his iterative compilation search space. He will write a small program that lists, for each point in the space, which loops should be unrolled and by how much. The unrolling decisions will be written to a file in some format. Now he will need to force the compiler to accept these decisions, reading from the file and overriding the default behaviour of the compiler. This will lead to code as sketched in figure 4.3; the necessary file parsing and searching has been elided for clarity but would require several lines of code, further obscuring the original heuristic.

The researcher also adds cycle counting to each function with an unrolled loop in it (a potentially non-trivial task) and runs all of his iterative compilations. The profiles gathered from each run have to be recorded in a database. This, undoubtedly, requires some effort. It is not, however, inside the heuristic, so we will come back to it later.

```
1  int decideUnrollTimes(loop* lp) {
2      int times;
3      if(shouldOverride) ...
4      else /*the heuristic*/
5      if(shouldPrintLoops) ...
6      if(shouldPrintFeatures) {
7          print(features_for_loop(lp));
8      }
9      return times;
10 }
```

Figure 4.4: Unrolling heuristic modified to print features about the loops.

4.2.4 Computing Features

After all the iterative compilation data is safely stored away in a database, the next step is to have some machine learning tool learn a model from that data so that it can predict for new, unseen loops what the best unroll factor should be. The researcher must have a list of features for every loop that the machine learning tool will base the model upon. He adds yet another block of code to the once clean heuristic as shown in figure 4.4. These features also need to be uploaded to the database for use by the machine learning tool.

4.2.5 Model Installation

Finally, armed with a predictive model, our researcher can embed it back into the compiler, enabling anyone to use the improved heuristic he has created. He adds more to the heuristic function; the machine learned version will replace the default on demand, as shown in figure 4.5.

The modified heuristic in figure 4.5 has elided practically all details of the changes and yet even then the original heuristic is dwarfed. The full version of the code would be large and the modifications made to enable just one particular machine learning experiment may exceed the size of the original. Different experiments may require still further changes. The new version is less maintainable than what went before and will likely not be shipped with any production version of the compiler. The researcher is left reimplementing his modifications every time the compiler is upgraded; a situation that is slow, error prone and deeply frustrating.

These issues and indeed several other, non-machine learning problems would all

```
int decideUnrollTimes(loop* lp) {
    int times;
    if(shouldUseML) {
        f = features_for_loop(lp);
        times = apply_model(f);
    }
    else if(shouldOverride) ...
    else /*the heuristic*/
    if(shouldPrintLoops) ...
    if(shouldPrintFeatures) ...
    return times;
}
```

Figure 4.5: Unrolling heuristic permitting a machine learned version to be used.

be solved if the compiler had been built to be extensible. Extensible software allows external users (in this case, our researcher) to adapt the behaviour of predefined points in the original software, all without changing a single line of the code in the original. A good extensibility library would permit heuristics to be replaced, reused or modified in clean fashion from outside the compiler. In our loop unrolling example, the researcher would have been able to use one extension to find out what loops his benchmarks have, another to force different loops to be unrolled according to his iterative compilation strategies and yet others to print loop features and to install the new heuristic. With extension capabilities the compiler would change from being an opaque black box to a fully customisable research compiler without compromising code quality, readability or maintainability.

The remainder of this chapter describes *libPlugin*, a powerful, feature rich, open-source extensibility library. Applied to GCC the compiler becomes perfect for machine learning research with minimal changes to the compiler's code. *libPlugin* is, in fact, completely independent of GCC, able to make any application extensible with very little work. However, the presentation here will focus on using the library for machine learning purposes and will, in particular, demonstrate how simple the above loop unrolling example becomes when supported by *libPlugin*.

4.3 *libPlugin*

libPlugin is an extensibility library for C which makes providing extension capabilities for applications easy. The library is application agnostic but was specifically constructed to make GCC extensible. One of the main goals of the project is that absolutely minimal changes should required to GCC to support extensibility. Often, when a fixed heuristic is converted to an extensible one, the differences are almost unnoticeable. Indeed, the changes to GCC amount to some ten lines of code in the main function to initialise *libPlugin* and often only one additional line of code per heuristic. The plug-in system is extremely simple to use without compromising power and flexibility.

Section 4.3.1 presents a brief overview of the system. The next section describes plug-ins, the main unit of abstraction in *libPlugin*. Then section 4.3.3 will explain the extension mechanisms. Section 4.3.4 briefly introduces *libPlugin*'s specific support for machine learning. More details on *libPlugin* are presented in appendix A.

4.3.1 Overview

The *libPlugin* environment consists of a set of interoperating components called plug-ins that can be loaded into the compiler at the user's request. Plug-ins are described in XML files and may additionally have shared libraries to implement some of their functionality. Plug-ins offer services to each other; the service is called an extension point and when another plug-in makes use of it that is called an extension. The information an extension point can require from extensions can be arbitrarily complicated, however, several convenient ways of constructing extension points take care of many of the common cases. In particular, several concepts from aspect orientated programming (*AOP*) (Kiczales et al., 1997) are offered, which are a good fit for machine learning purposes.

For example, *libPlugin* provides a plug-in to control loop unrolling. The plug-in allows other plug-ins to replace the unrolling heuristic with their own C function. They might also register C functions to listen to events informing them about unrolling decisions. The other plug-ins would package these C functions in shared libraries and inform *libPlugin* about them with XML specification files. The unrolling plug-in offers the original heuristic as an AOP style *join-point* for full flexibility. However, *libPlugin* does not force such low-level access. The unrolling plug-in also offers a simplified interface, wherein the user can merely list the unroll factors individually for each loop. Moreover, details of the unrolling process can be recorded easily. All of this

is managed by the same coherent specification system which *libPlugin* will administer for the user.

libPlugin was inspired by Eclipse, the extremely successful plug-in enabled integrated development environment for Java. Aspect orientation was inspired by AspectJ, an AOP language targeted at the Java virtual machine.

4.3.2 Plug-ins

In *libPlugin*, software is arranged into components called *plug-ins*. A plug-in is similar to a module in other languages; it provides a collection of related services to and uses the services of other plug-ins. Some plug-ins are provided with the compiler, representing its core services, others are installed optionally by the user.

The services that a plug-in provides are called *extension points*, which will be discussed further in section 4.3.3. When one plug-in uses another's extension point it creates an *extension*. An extension point is an arbitrary service; *libPlugin* places no restrictions on how complex or powerful the service might be. It does, however, offer many convenient ways to easily create common types of extension point so that the compiler writer can quickly make heuristics extensible.

A plug-in may have one or more shared libraries that implement its services. These libraries are loaded only if the plug-in is loaded (see below), reducing bloat in the runtime footprint. Not all plug-ins need the power of shared-libraries to do their work, however; many useful things can be accomplished without a single line of C code just by extending other plug-ins. Examples of creating plug-ins are given in appendix A.

A pictorial representation of GCC with plug-ins is shown in figure 4.6. In the figure are several of the core plug-ins (represented as being embedded in GCC), some of which also have supplemental shared libraries which implement additional, related services that are not present in the core compiler. Other, user supplied plug-ins are visible which may also provide their own extension points.

All plug-ins are visible to the compiler when it starts through a set of search paths. The compiler, however, will only load a few of the plug-ins. Some will be specified as *eager* plug-ins which are always to be loaded; an example being a new optimisation or compiler pass that the user wants to always be active. Other, *lazy* plug-ins, will be loaded only when required. All of the plug-ins provided with GCC are lazy, so unless the user requests a plug-in to be loaded, there will be no change from the default compiler behaviour. The compiler user can trigger the loading of lazy plug-ins with either

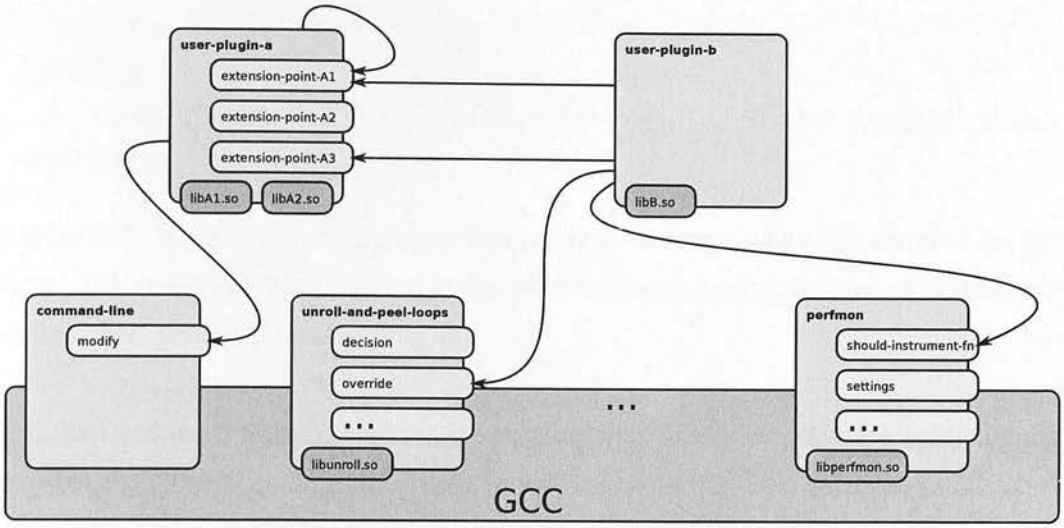


Figure 4.6: Plug-in and extension point interaction. Here a few of GCC's core plug-ins are shown with their implementation shared libraries and the extension points they offer. Other, user plug-ins also offer extension points. Plug-ins can extend extension points (shown as arrows in the figure). A plug-in can even extend its own extension points. Extension points can be extended multiple times by different (or the same) plug-ins.

command line arguments when they invoke the compiler or by setting environment variables.

Once the initial set of eager and user requested plug-ins is known, *libPlugin* will begin to load them. If a loaded plug-in requires the services of an unloaded plug-in, that plug-in will also be loaded. The system then recursively loads all the plug-ins it needs and no more.

Many software systems allow user supplied plug-ins to extend the application's default behaviour. Some of these do not regard plug-ins as highly as the core application. For example, they may allow plug-ins to extend some aspects of the application, but not allow plug-ins to offer their own extension points to other plug-ins; plug-ins, in those cases, are second class citizens of those applications. *libPlugin*, on the other hand, promotes a cooperating ecosystem of plug-ins; user plug-ins are just as powerful as core plug-ins.

4.3.2.1 Plug-in File Format

For each plug-in there is an XML description file. This file tells *libPlugin* everything it needs to know about the plug-in, from its dependencies to the extension-points it

```
1 <?gcc version="4.3"?>
2 <plugin id="hello-world">
3     <extension point="message.start">Hello, World!</extension>
4 </plugin>
```

Figure 4.7: A `hello-world` plug-in that prints a message when the compilation begins. The message is embedded in the specification, showing that much can be accomplished without C code.

provides and uses. Some plug-ins need no more than this description file, while others might need some C code to drive their functionality. In the latter case, the plug-in has some number of shared libraries in addition to the XML description.

The minimal plug-in specification is shown in figure 4.7. The plug-in will cause GCC to print `Hello, World!` to the standard output when it compiles a file. This XML plug-in specification is either placed on a special plug-in search path or mentioned on the command line to GCC. Each plug-in specification file must contain valid XML indicating what applications it should work with and contain a valid `<plugin/>` element. In the simple `hello-world` example, line 1, declares that the plug-in applies to GCC with at least version 4.3. In line 2, the plug-in gives itself an identifier and, lacking a command to force eager loading, that it is, by default, a lazy plug-in. Plug-ins use identifiers to declare that they depend upon each other and users also give these identifiers to load optional plug-ins from the command line. Plug-ins can, in fact, be anonymous, but it is considered good practice to always name them. Line 3 says that this plug-in extends another plug-ins extension point. The identifier of that point is `message.start`. *libPlugin* will ensure that the plug-in providing that extension point is loaded and that only one such plug-in exists. In this case, the extension point is simple, and happens to be provided by the `message` plug-in. It will print the text contents of the extension to the standard output. Line 4 ends the specification.

4.3.3 Extension Points and Extensions

Plug-ins bundle together related functionality into sensible units. The majority of the work, however, is performed at a finer grained level, that of extension points. At its most simple an extension point is simply a named object which has an `extend` function with which other plug-ins can pass it snippets of XML. The XML can be arbitrarily complicated and include pointers to symbols from the extending plug-in's

```
<extension point="gcc-rtl-unroll-and-peel-loops.override">
    <loop
        main-input-file="foo.c" function="bar" loop="2"
        times="10"/>
</extension>
```

Figure 4.8: An extension for point `gcc-rtl-unroll-and-peel-loops.override`, specifying unrolling settings for a single loop.

shared libraries, so there is no limit to the power of the extension mechanism.

For example, in plug-in `gcc-rtl-unroll-and-peel-loops`, GCC provides an extension point, `gcc-rtl-unroll-and-peel-loops.override`, which allows other plug-ins to override the default loop unrolling heuristic for any loops it chooses. If a plug-in includes the snippet shown in figure 4.8 in its specification file, it will extend that extension point, asking for loop two in function `bar` from file `foo.c` to be unrolled 10 times.

The implementation of the extension point itself is in two parts. First there must be some C function to accept any extensions. In pseudo-code it looks something like figure 4.9. The extension function remembers what overrides it is given. It also needs to replace the default unroll heuristic with something that will use the overrides when given. In fact, the unrolling heuristic is represented by another extension point so it can be programmatically overridden. This demonstrates one of the powerful aspects of the extension point system; the ability to compose different extension points to give layers of functionality. The first extension point allows low-level alterations to the heuristic and another gives a simpler, high-level but less flexible wrapper.

The plug-in must also declare the `gcc-rtl-unroll-and-peel-loops.override` extension point. It does this by putting the contents of figure 4.10 in its XML specification. This declaration informs *libPlugin* that whenever another plug-in extends the extension point the `overrideExtend` function from its shared library should be called.

Although the extension mechanism is very simple to arrange it does require some coding of the extension function which invariably involves an amount of tedious XML processing[1]. Since one of *libPlugin*'s goals is to make extensibility as simple as possible, it offers a number of shortcuts for defining powerful extensions for the most common cases with almost no code. The following sections describe the easy ways to create and use convenience extension points.

[1] *libPlugin* is built on top of the open source `libXML2` library.

```
list overrides = NULL;

bool overrideExtend(
    ExtensionPoint* self,
    Plugin* extender,
    xmlNodePtr specification
) {
    replace unroll heuristic with overrideUnroll;
    for each child in specification {
        append child to overrides;
    }
    return TRUE;
}

int overrideUnroll( loop* lp ) {
    spec = first element in overrides matching lp;
    if( spec == NULL ) return previous heuristic;
    else return spec.times;
}
```

Figure 4.9: Pseudo code to implement an extension point which supports simplified unrolling specifications. The code here is for the shared library, written in C.
First a list is created to remember each time another plug-in extends the extension point. When another plug-in does extend the point, `overrideExtend` will be called with that plug-in's details and the XML of the extension. That function records all relevant unrolling commands it finds in that XML. Finally is `overrideUnroll`, a replacement for the default heuristic, which scans the list for matching commands, deferring to the original heuristic if necessary.

```
<extension-point id="gcc-rtl-unroll-and-peel-loops.override">
    <extend symbol="overrideExtend"/>
</extension-point>
```

Figure 4.10: XML specification for the extension point which supports simplified unrolling overriding. The plug-in need only name the extension point and give the symbol for the extension function from figure 4.9.

Events Events are a common programming pattern that are extremely useful in an extensible compiler. Consider, for example, the case when a user would like to know what loops have been unrolled and with what unroll factor. He might choose to log this information to a file or to a database or to aggregate it in some other way as the loops are unrolled. If the compiler fires an event every time it unrolls a loop, then users can listen to those events and do anything they want. The compiler is thereafter free from worrying about whether is has supported every possible user interaction.

libPlugin makes creating, firing and listening to events trivial. The compiler writer declares a function pointer which initially points to an empty function and tells *libPlugin* that the function pointer is an event. If another plug-in wants to listen to the event, *libPlugin* will replace with function pointer with dynamically built code that will inform any listeners that the event fired. The compiler writer can then just call the function pointer whenever they want to fire the event. The cost of this extensibility is only one additional indirection when no listeners are applied, making *libPlugin* a very efficient system. More details and an example of the event mechanisms are provided in appendix A.

Around Advice One of the primary needs of machine learning in compilers is to be able to replace the default behaviour of an heuristic. To support this, *libPlugin* borrows a concept from aspect orientated programming (AOP). AOP allows developers to add *advice* to methods that have already been written (Kiczales et al., 1997). One form of this advice replaces the method with a new one which receives the same original arguments. The advice can perform any operation it desires but in particular can also, if it needs to, call the original method it replaced. In fact, these advices can be layered, with one method being advised multiple times, and each layer of advice being able to call the next one down. The AOP formulation has been very successful and has been demonstrated in a large number of real world projects; it is also a perfect fit for the machine learning in compilers requirements.

libPlugin allows plug-in writers to specify that a function can be advised and for plug-ins to advise that function. The formulation is very similar to that of events; the compiler writer simply indirects all calls to the heuristic function via a function pointer. *libPlugin* will then replace that pointer if any other plug-in wishes to override the heuristic. Generally, around extension points are created only as part of the more powerful join points which are described next. Again, more details are in the appendix.

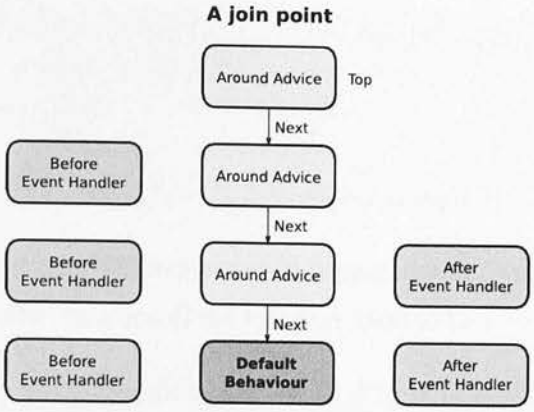


Figure 4.11: Anatomy of a join point.

Join-Points Around extension points allow functions and heuristics to be modified. They have some small practicality limitations, however. A typical usage pattern for altering heuristics is that we may wish to receive an event when the function is first called with the arguments passed to it and another when the function terminates, this time with the return value as well as the arguments. These events are useful when some kind of reporting is required which does not override the behaviour of the advised function. We cannot simply have some around advice which performs the logging and then delegates to the next advice on the stack without altering arguments or return value since we cannot guarantee the ordering on the advice stack.

AOP solves this problem with a concept called a *join point* (Kiczales et al., 1997) and *libPlugin* borrows that concept. A join point consists of exactly the two event extension points and an around extension point that we need as shown in figure 4.11. When a join point is called, first all of the before event listeners are notified with the function's parameters. Then the top advice on the around advice stack is called with those parameters, which may or may not call further down the advice stack. Finally the after event listeners are notified with the return value from the top advice and the original parameters of the function call.

Creating and using a join point in *libPlugin* is very simple. For example, if the C code for the loop-unrolling heuristic is originally as in the beginning of the motivating example from figure 4.1 then we can turn it into an extendible join point by converting it to a function pointer instead. This requires only one additional line of code and all uses of the function remain exactly as they were; the compiler is not cluttered with ugly extensibility code. The heuristic now looks like figure 4.12.

The join point needs to be declared to *libPlugin* in a plug-in XML specification

```
static int decideUnrollTimes_original(loop* lp) {
    int times = /*the heuristic*/;
    return times;
}
int (*decideUnrollTimes)(loop* lp) = decideUnrollTimes_original;
```

Figure 4.12: Unrolling heuristic indirected in preparation for converting it into a join-point. After the indirection no uses of the function need to be changed.

```
<join-point id="decide-unroll-times" signature="int f(loop*)">
    <call symbol="decideUnrollTimes"/>
</join-point>
```

Figure 4.13: Plug-in specification for a simple unrolling join-point.

file, giving an identifier for the join point, the prototype of the function so that dynamic code can built for it and the symbol name of the function pointer to be replaced:

The join point is not in itself an extension point. Instead, a join point creates three extension points. If the identifier of the join point is `x`, then the first event extension point will have identifier, `x.before`, the around extension point will have `x.around` and the last event will have `x.after`. These extension points are then used as normal.

If no plug-in listens to one of the join point's events or places advice on the around stack then the function pointer, `decideUnrollTimes`, will still point to its original value, the function `decideUnrollTimes_original`. Only if necessary is any dynamic code constructed and the function pointer updated. In this way, just as for events and around extension points, there is practically no cost to making the compiler extensible.

List The simplest type of convenience extension point provided by the system is a list of values. The null-terminated list can contain a pointer to any type of data and other plug-ins can append values to the list.

For primitive types (integers, floating point numbers and strings) the element to be appended can be given directly in the extending plug-in's XML description. More complex types are handles by either simply providing a symbol which points to the element in a shared library or by giving a factory method that interprets the XML specification to create the list element.

Hook A hook allows a function's implementation in one plug-in to be replaced by another plug-in. A hook in the owning plug-in is simply a function pointer which

another plug-in can overwrite. The owning plug-in can then call the hook whenever it needs to. Only one plug-in can extend a hook; the system will report an error if two plug-ins attempt to extend the same hook.

Hooks are extremely limited; for nearly all cases the join-point extension (see above), which is much more powerful should be used instead. However, there is a cost to join-points because no dynamic binding code needs to be generated (this cost is only paid when the join-point is extended). Hooks, on the other hand, provide extensibility at only the cost of an indirect function call. They should be used only when the hook will be called so often that the greatest efficiency is required.

4.3.4 Machine Learning Plug-ins

libPlugin for GCC comes with a number of plug-ins that are useful for machine learning tasks. This section lists those plug-ins with a short description of each. More details are provided in appendix A.

4.3.4.1 Heuristic Control and Reporting Plug-ins

These plug-ins allow heuristics to be overridden. They provide mechanisms to set the heuristic choices with simple XML files or by programmatic replacement. In addition they offer reporting capabilities so the researchers can discover information about the potential heuristic choices and the actual choices that are made.

Command Line Many iterative compilation and machine learning tools search over compiler flags. While it would be possible to alter the makefiles of every project involved, it is error prone and difficult to automate. *libPlugin* has a plug-in to help which allows the compiler's command line arguments to be specified regardless of the makefile. Users can remove and insert command line arguments at will, using either a simple, pattern matching specification or programmatically.

Loop Unrolling This plug-in allows loop unrolling factors to be specified for each compiled loop, either programmatically or via a simple XML format. It supports both iterative compilation with an outside driver program or heuristic replacement. The plug-in also gives the opportunity to record information about which loops were unrollable and which were unrolled.

Auto-Vectorisation GCC has an optimisation to which tries to automatically vectorise loops. *libPlugin* provides a plug-in to print information about vectorisation operations, to query various vectorisation aspects of loops and to control which loops are vectorised. It is a very similar plug-in to the loop unrolling plug-in. The essential difference is that rather than specifying the number of times the loop should be unrolled, the target is given. The target is the processing unit that the code will be vectorised onto (e.g. not at all, a SIMD unit, the main CPU, etc.)

Inlining Function inlining replaces a function call site with the body of the function being called. There is a plug-in in *libPlugin* which allows control of inlining. Again, the plug-in provides services to print information about inlining, query inlining characteristics (such as whether a call site is inlinable) and to control which call sites are inlined.

Pass Manager Passes[2] in GCC are where it performs the majority of its work. The compiler contains some 180 different passes which transform the code toward the machine level and apply all of the numerous optimisations that GCC supports. *libPlugin* comes with a plug-in which supports machine learning experiments at the pass re-ordering level as well as enabling compiler extensions that add new passes. The plug-in permits passes to be forcibly turned on or off, overriding their gate functions. It also allows complete reordering control over the pass tree on a per function basis, either through a simple XML specification or programmatically.

There is also a plug-in which will log a list of all the passes each function is run through while being compiled. The log can be sent to various destinations, including a database, and in several formats.

4.3.4.2 Features

Machine learning experiments require features to be given to the model learner and, once a model has been created, the model uses them to compute its predications. There are many different features that can be thought of for any code block. Researchers often have different needs, some will ask for features at the basic block level, some only on functions, others on instructions. Even if targeting the same level of code element, researchers will come up with different features.

[2]Other compilers may use the term 'phase' for GCC's 'pass'.

libPlugin comes some ten or so plug-ins that compute features for different code elements. Most have been created to the specifications of other researchers. Two of the feature sets, Stephenson's and Milepost's, are well known having been reimplemented from other papers; they are described below. One of the more simple feature plug-ins, built for a colleague, is also described. All the feature plug-ins have a very similar interface and similar capabilities; for all feature sets, the output format and destination can be configured as well as a set of pattern matching filters to select the functions of interest.

Stephenson's Features Stephenson(Stephenson and Amarasinghe, 2005) showed that machine learning could successfully outperform the heuristics built by human experts. His experiments targeted loop unrolling and so he needed features capable of describing loops to the machine learning tools. *libPlugin* has an implementation of these features. The full list of Stephenson's features is given in table 6.10.

Milepost Features The Milepost(Fursin et al., 2008b) project produced a standard set of features given in table A.3. In that project, the features are produced by writing out the whole AST to a Datalog database and then each feature is represented as a Datalog relation over that database. The features are at the function level only.

libPlugin has a plug-in which reimplements the Milepost features. The implementation is simpler, just written in C, and considerably faster than the original Datalog implementation.

Instruction Count Features This plug-in will create one feature for each type of AST or RTL node in each basic block. An instruction count histogram for each basic block is written to a file or a database.

Printing the Intermediate Representation *libPlugin* allows the full AST or RTL data of functions to be printed to files. This is heavily used by the automatic feature generation techniques that are presented in chapters 5 and 6 to generate source data.

The plug-in prints the internal representation of functions in a convenient XML format. The information is sent only to files, not a database, simply because of the volume of data produced, which may be several gigabytes for some benchmarks. The print is configured in much the same ways as the feature plug-ins; i.e. the output file can be specified and the events that cause the print can be given, too, with the same

syntax.

4.3.4.3 Instrumentation Plug-ins

These plug-ins transform code as it is being compiled to apply additional instrumentation that GCC does not natively support.

Perform A plug-in allows GCC to instrument functions it is compiling so that when run they will count the number of cycles spent in the function. Cycle counts are very useful when performing iterative compilation on functions or smaller code elements as they give timings to very high precision. Other techniques might be too coarse grained to time the small elements. Users can specify which functions should be instrumented and where and how to output the results, be it to a file or database. Users can make these settings programmatically or via a simple, XML data format.

Trace The tracing plug-in causes GCC to instrument the files it is compiling to create traces of each basic block as it is executed. These traces create very large data files but may offer useful information to machine learning tools.

4.4 Motivating Example Reprise

The previous sections showed how *libPlugin* makes the compiler easily extensible and introduced a number of useful plug-ins for machine learning in GCC. We can now revisit the motivating example from earlier in this chapter and discover how the same task would be done using *libPlugin*. The researcher will not need to alter a single line of GCC and will be able to distribute his updated heuristic to others who will also not need to change GCC to use it. Recall that in the example our researcher wanted to learn a new model to predict the best loop unrolling heuristics.

4.4.1 Plug-in Enabling the Original Heuristic

A small change is needed to the original heuristic to allow *libPlugin* to manage it. The compiler writer performs these changes, not the researcher using GCC. The researcher makes no changes to GCC at all. The only change the compiler writer needs to make (other than initialising the *libPlugin* library in the compiler's main function) is to indirect calls to that function via a pointer as shown in figure 4.14.

```
1 static int decideUnrollTimes_original(loop* lp) {
2     int times = /*the heuristic*/;
3     return times;
4 }
5 int (*decideUnrollTimes)(loop* lp) = decideUnrollTimes_original;
```

Figure 4.14: The loop unrolling heuristic indirected to become plug-in enabled.

This is the only noticeable change to GCC's source code; the heuristic is now completely exposed to users. The compiler writer also writes a small plug-in specification file describing the name and signature of the heuristic function, declaring it to be a join point. He also may provide additional plug-ins which make interacting with the heuristic simple (several are supplied with *libPlugin*). Those additional plug-ins, however, also do not require any changes to the compiler's source. Due to *libPlugin*, all conceivable extensions to the heuristic can be made without any reduction in the readability and maintainability of the compiler's source code.

4.4.2 Determining Unrollable Loops

Our researcher's first task was to find what loops each benchmark contains. Ideally he would also get the range of unroll factors that can work for each loop. This task is easy enough, he simply loads plug-in, `gcc-print-rtl-unroll-and-peel-loops`, when compiling his code. He will also have to ensure that loop unrolling is enabled (it is not by default) which can be done by putting `-O3` on the command line, making sure that no other conflicting flags are there. Before *libPlugin* our researcher had to understand and alter all of the makefiles and build scripts of all of his benchmarks.

Now he writes a small plug-in which loads the loop printing plug-in and changes the command line in one go, shown in figure 4.15. This plug-in is an XML file that the researcher drops into *libPlugin*'s search path. Line 1 indicates that the plug-in is for GCC and line 2 gives the plug-in a name. Lines 3 to 6 first remove all command line arguments that start with `-O` and any `-fno-unroll-loops`, before adding `-O3` to the command line. Finally, line 8 invokes a plug-in that prints information about the loops in the program and that will also record what the default unrolling heuristic would choose to do for those loops.

When the compiler is run with this plug-in in its search path, there will be a file in the current directory, `gcc-print-rtl-unroll-and-peel-loops.log.xml`. The file will contain entries like those shown in figure 4.16. These entries tell the researcher

```
1 <?gcc version="4.3"?>
2 <plugin id="find-unrollable-loops">
3     <extension point="command-line.modify">
4         <remove><arg>-O*</arg></remove>
5         <remove><arg>-fno-unroll-loops</arg></remove>
6         <insert><arg>-O3</arg></insert>
7     </extension>
8     <requires plugin="gcc-print-rtl-unroll-and-peel-loops"/>
9 </plugin>
```

Figure 4.15: A plug-in to print information about unrollable loops. Lines 3 to 7 ensure that loop unrolling is enabled and line 8 loads a plug-in that will print the relevant information.

```
1 <loop main-input-file="foo.c" function="bar" number="1">
2     <unrollable type="simple"/>
3     <unrollable type="stupid"/>
4     ...
5     <unrolled type="simple" times="8"/>
6 </loop>
```

Figure 4.16: Example information describing the unrollable loops, generated by plugin `gcc-print-rtl-unroll-and-peel-loops`.

```
1  <?gcc version="4.3"?>
2  <plugin id="<benchmarkname>-1">
3      <!-- Still have to turn on unrolling -->
4      <extension point="command-line.modify">...</extension>
5
6      <!-- Select the unrollings -->
7      <extension point="gcc-rtl-unroll-and-peel-loops.override">
8          <!-- Start with all loops not unrolled -->
9          <loop times="0"/>
10         <!-- Unroll one loop per function -->
11         <loop main-input-file="foo.c" function="bar" number="2" times="2"/>
12         ...
13     </extension>
14
15     <!-- Get cycle counts for each function -->
16     <extension point="gcc-perfmon.settings">
17         <output db="couchdb" host="server" tag="{plugin.id}"/>
18     </extension>
19 </plugin>
```

Figure 4.17: Plug-in to unroll particular loops in a benchmark and generate cycle counts. Lines 7 to 13 describe the unrolling factors and lines 16 to 18 insert the profiling instrumentation.

which loops are in the benchmark, whether they can be unrolled with different flavours and how the default heuristic was applied. He could have had this data sent directly to a database with only one additional line in the XML, describing the location of the database. Compared to the work previously required, our researcher has had to do very little and, again, GCC has not been changed in any way.

4.4.3 Iterative Compilation

Now that he has a list of unrollable loops for each benchmark, the researcher can begin his iterative compilation. He writes a program to parse the unrollable loops file and spit out a plug-in for each point in the iterative compilation search space. This plug-in needs to unroll some loops and add cycle counting to every function. Each of those plug-ins will look like the example in figure 4.17. Lines 1 to 4 are similar to the previous plug-in and perform much the same role. Lines 7 to 13 specify how to unroll

```
1 <?gcc version="4.3"?>
2 <plugin id="get-features">
3     <extension point="gcc-features-loop-stephenson">
4         <output db="couchdb" host="server" tag="<benchmarkname>"/>
5     </extension>
6 </plugin>
```

Figure 4.18: Plug-in to store static loop features into a database.

each loop. Line 9 states that all loops not later specified will not be unrolled. Line 11 onwards force an unrolling factor for each individual loop. The researcher could have easily wrapped the heuristic with his own C code, still without altering GCC, but since *libPlugin* already allows convenient override capabilities he chooses that path. Lines 16 to 18 do something that was not deeply considered in the original motivating example; it instruments each function to record cycle counts. Now every time the benchmarks are run, the profiling data will be recorded to the database. Compared to life before *libPlugin* this is an enormous improvement. The researcher compiles each point in the space and runs the resulting programs.

4.4.4 Computing Features

With the iterative compilation done, the researcher needs to get features for each loop. He decides to use Stephenson's features (Stephenson and Amarasinghe, 2005) to start with, compiling each benchmark with the plug-in shown in figure 4.18. Again, this is all that is required. Stephenson's features will be uploaded automatically to the researcher's database for every loop in his programs. He could choose to use different features instead (several such plug-ins are included in *libPlugin*) or easily create his own with some custom C code. Once more, he has not had to alter GCC.

4.4.5 Model Installation

The researcher now has all the information he needs stored away in his database. The effort required to achieve this was minimal compared to doing it without *libPlugin* and no changes had to be made to the compiler. He can now scan through the database and learn a model which predicts the best unroll factor for each loop. At this point he will write a plug-in (finally needing some C code, although the plan for future versions is to directly support a number of standard machine learning techniques) to insert his model

```
int machinelearning_unrollTimes(loop* lp) {
    StephensonFeatures features;
    StephensonFeatures_compute(&features, lp);
    return /*Do ML stuff with features*/;
}
```

Figure 4.19: Plug-in C code to install the model in GCC.

```
<?gcc version="4.3"?>
<plugin id="machine-learning-unrolling" lazy="true">
    <library path="mlunroll.so"/>
    <extension point="gcc-rtl-unroll-and-peel-loops.decision.around">
        <callback symbol="machinelearning_unrollTimes"/>
    </extension>
</plugin>
```

Figure 4.20

into GCC. The C code is shown in figure 4.19. This function will need to be compiled as a shared library, but *libPlugin* provides tools to seed such projects easily and thus the researcher will not even need to write a makefile for such a simple purpose.

He can insert this into GCC with a small plug-in specification file, shown in figure 4.20. The plug-in loads the shared library created for the C code of figure 4.19 and replaces the default unrolling heuristic with the machine learned one as required. Users of this plug-in will explicitly invoke it since it is a lazy plug-in and will hence not be loaded by default.

The new plug-in can be packaged up and sent to any interested researcher who uses *libPlugin* (again, packaging tools are provided). No users will need to change the compiler to try the new machine learning plug-in.

4.5 Summary

This chapter has shown how difficult and error prone it is to perform machine learning experiments in modern compilers. It has introduced *libPlugin*, an extensibility framework that solves these problems. *libPlugin* allows researchers to do experiments without having to alter GCC's source code and to share their results in a cooperative, modular fashion. It supports modern software engineering practice. In addition, *libPlugin* comes with a number of predefined tools that make common machine learning tasks trivial.

Chapter 5

Feature Grammars

Recent work has shown that machine learning can automate and in some cases outperform hand crafted compiler optimizations. Central to such an approach is that machine learning techniques typically rely upon summaries or *features* of the program. The quality of these features is critical to the accuracy of the resulting machine learned algorithm; no machine learning method will work well with poorly chosen features. However, due to the size and complexity of programs, theoretically there are an infinite number of potential features to choose from. The compiler writer now has to expend effort in choosing the best features from this space. A novel mechanism is developed to automatically find those features which most improve the quality of the machine learned heuristic. The feature space is described by a grammar and is then searched with genetic programming and predictive modelling.

This chapter describes how we design grammars to define the space of features, while the next explains how the feature space is searched for good features. This chapter is organised as follows. Section 5.2 delves deeper into the problems of manually written features. Section 5.3 explains how a feature space is formulated. Then section 5.4 shows how features can be generated from the feature grammar and what problems are encountered during this process. Section 5.5 outlines the support required to actively search the feature space before concluding the chapter.

5.1 Introduction

Supervised machine learning needs features to work and to date, whenever machine learning has been applied to compilers, the features have been hand written by a human compiler expert. The expert wants to replace some heuristic with a machine learned

one and finds all the data structures available in the compiler when that heuristic is computed. From these data structures he will need to compute his features. The problem is that because the data structures are mostly trees and graphs of unbounded size, there are an infinite number of possible summaries he can choose from for features. The expert can only try a few and each takes some effort to implement.

Not only is he faced by countless features to choose from, but features that are based on human intuition may not be the most successful because the interaction between features and a machine learning algorithm is complex. If the features do not represent all of the relationship between the program and the desired outcome they may not work sufficiently well with the machine learning algorithm. No machine learning tool will create quality predictions for new programs if there is little to learn from the input examples. In some ways the use of machine learning has pushed the problem from one of hand-coding the right heuristic to one of hand-coding the right features.

Previously, researchers in machine learning for compilers have manually created lists of features they believe reasonable. Many such works use feature selection to remove redundant or unhelpful features. However, none have attempted to search through the feature space, generating entirely new features along the way or even acknowledged the existence of the space itself. In this work, on the other hand, an automated system is allowed to search through an infinite feature space to find features which most improve the machine learning algorithm's performance. In this approach the space of features is represented as a grammar where each sentence from the grammar represents one feature. The human is at last relieved from deciding which features are important and which are not.

The main contributions of this chapter are the development of grammar based system to describe a space of features and of search mechanisms which combine the best of previous approaches without inheriting their drawbacks.

5.2 Manual Feature Creation

This section describes some of the problems that the compiler writer must be aware of when writing features by hand. Then the way features are used in previous work is shown to fail to help a machine learning tool to make a good prediction.

5.2.1 Difficulties with Human Created Features

Irrelevant features It is easy to conceive of features that will have no relevance to any optimisation task. Features such as 'the number of comments in the code' or 'the average length of identifiers' would not help a machine learning algorithm. These examples are so obvious that no human would suggest them but the same situation arises in more subtle cases where a sensible sounding feature simply has no bearing on the current optimisation.

More serious is the case when a feature is useful on its own but when added to an existing set of features does not show any additional improvement. This can happen when the all of the useful information in a feature is already present in the other features, for example if the feature is a linear combination of the others.

The compiler writer can somewhat mitigate the damage of irrelevant features by applying feature selection to weed out the unnecessary features. However, the features may also introduce noise which selection algorithms will fail to remove and which can mislead the machine learning tool.

Classification clashes Two distinct programs may have the same feature vector but different best values for the heuristic; a machine learning algorithm will predict at least one of them wrongly. This does in fact happen in practice as shown in Monsifrot et al. (2002). The presence of these clashes is a clear indication that the features are inadequate since they cannot distinguish examples from different classes. However, irrelevant features and noise can obscure classification clashes.

When clashes are found they place an upper bound on the accuracy of the models created by the machine learning tool. It is possible that adding other features may help to separate the features by adding other dimensions.

Classifier peculiarities A set of features that performs well for one machine learning algorithm might not be good for another (Kohavi and John (1997)). In other words features are not independent of the learning technology.

Beyond simple features Once the 'obvious' features have been written, inevitably they do not completely represent the relationship between the programs and the desired heuristic values. The expert must then choose which additional features to implement. The sheer number of choices can be huge and the expert will find that each stage, as they increase the complexity of their features they face the same difficult challenges

as they did before, only now multiplied due the larger set of features they must work with.

5.2.2 Motivating Example

```
1 for (i=0;i<EXP_TABLE_SIZE-1;i++) {
2     l->SpotExpTable[i][1] =
3         l->SpotExpTable[i+1][0] -
4         l->SpotExpTable[i][0];
5 }
```

(a)

Method	Unroll	Cycles	Speedup	% of Max
Baseline	0	406,424	1.0000	0%
Oracle	11	328,352	1.2378	100%
GCC Default	7	418,464	0.9712	-12%
GCC Tree	2	392,655	1.0351	14%

(b)

Figure 5.1: Loop from MediaBench (a) and speedups using various schemes (b). GCC's default heuristic selects an unroll factor of 7 causing a slowdown. Using GCC features and machine learning, an unroll factor 2 is selected giving a small improvement.

This section demonstrates that selecting the right features can have significant impact on optimisation performance. Consider the loop in figure 5.1 selected from the `mesa` benchmark within *MediaBench*. If GCC's default loop unroll heuristic (labeled GCC Default in figure 5.1 (b)) is applied, it determines the best unroll factor is 7. When executed on the Pentium this achieves a slowdown of 0.97. However, if all loop unroll factors up to 15 are exhaustively evaluated, then the best unroll factor is found to be 11 resulting in a speedup of 1.24 as shown by the Oracle entry in figure 5.1 (b). If GCC's heuristic is replaced with a machine learning decision tree algorithm, whose features are the same information used by GCC's heuristic (as shown in figure 5.2 (a)), then it is possible to achieve a speedup of 1.04 or 14% of the maximum available. Figure 5.2(b) shows the path followed by the learned decision tree heuristic leading to the unroll factor of 2 being selected. It will be discovered in the next chapter, in section 6.3,

that by automatically searching a feature space, features which allow the full speed up, 100% of the maximum, can be found.

Name	Value
ninsns	10
av_ninsns	9
niter	6.14E17
expected_loop_iterations	49
num_loop_branches	1
simple_p	1

(a)

```
if( ninsns <= 63 )
 if( simple_p > 0 )
   if( num_loop_branches <= 3 )
     if( av_ninsns > 5 )
       if( niter > 6.1384926724882432E17 )
         if( expected_loop_iterations > 8 )
           if( niter <= 6.1428835034542899E17 )
             if( num_loop_branches <= 1 )
               unrollFactor = 2;
```

(b)

Figure 5.2: The path through the learned GCC tree heuristic (b) for the example in figure 5.1 and the features used in that path (a). The features are the variables that GCC's original heuristic examined when deciding its value.

5.3 Defining the Feature Space

This section explains how feature grammars are used to describe a feature space. A toy compiler language is presented first in section 5.3.1 and features are defined for it. How features are computed against the compiler's IR is covered in section 5.3.2. Implementation of advanced feature spaces that are difficult to define with a context free grammar alone is covered in 5.3.3. Then, section 5.3.4 shows how different areas of the feature space can be prioritised over others.

The probabilistic context free grammars have been used before to generate program fragments (Ryan et al., 1998). However, they have never been used to generate machine

```
1 <expr> ::= <term> <op> <expr>
2 <term> ::= <id> | <num> | "(" <expr> ")"
3 <op>   ::= "+" | "*"
4 <id>   ::= ("a" | ... | "z")+
5 <num>  ::= ("0" | ... | "9")+
```

- a = 10
- b = 20
- c = a * b + 12
- d = a * ((b + c * c) * (2 + 3))

Figure 5.3: A grammar for the simple language and example statements from it.

learning features for compilers.

5.3.1 Features for a simple language

A toy example is presented to show how the process works. The toy language allows only sets of assignment statements; the left hand side of each will be a variable name; the right will be an expression containing variables, constant integers, operators '+' and '*' and parentheses. How to parse this language is of no concern, therefore it is assumed that the ambiguity in operator precedence has been suitably dealt with. Only the intermediate parse trees are interesting. A BNF for the simple language and example statements are shown in figure 5.3.

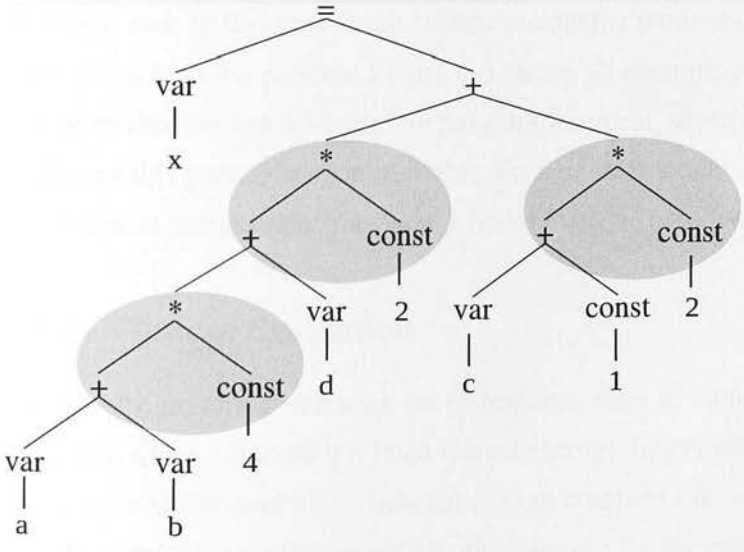
A human compiler expert, when thinking of features to be computed over expressions will most likely devise simple features like, *the number of '*' operators in the expression* or *the depth of the expression.* He will implement and test these features and with luck will discover that they are somewhat helpful to machine learning but in all likelihood do not perform as well as he had hoped. He will then, probably, create small variations of his original features. For example, to expand on his count of multiply nodes he may think that another feature could be to count those multiply nodes which have as their left child a '+' operator and whose right child is constant. He can add this feature to his set and try his machine learning tool again.

There are an infinite number of these features, however, as the compiler expert may choose to further refine his new feature by specifying the properties, recursively, for

```
<feature> ::= "countNodesMatching(" <matches> ")"
<matches> ::= "isConstant" | "isVariable" | "isAnyType"
            | ("isPlus" | "isTimes")
              ("&& leftChildMatches(" <matches> ")")?
              ("&& rightChildMatches(" <matches> ")")?
```

Figure 5.4: A simple feature grammar. Sentences from the grammar are expressions which can be evaluated over statements from the toy language. Each feature counts the number of sub-trees in the toy language statement which match a pattern.

```
countNodesMatching(
    isTimes &&
    leftChildMatches(
        isPlus
    ) &&
    rightChildMatches(
        isConstant
    )
)
```



x = ((a + b) * 4 + d) * 2 + (c + 1) * 2

Figure 5.5: An example feature from the grammar in figure 5.4. In the right hand of the figure is a sample AST from the tiny language, showing the matching sub structures.

The feature thus evaluates to three.

the children of the '+' node; for example restricting the set of values for the constant or any number of complex modifications that could be dreamed up. The expert will repeat this process, continually implementing and testing more features until either no further improvement in the machine learned model is found or he runs out of time to experiment.

The approach taken here is an automation of this labour intensive process. Since this system will explore the space of potential features, it must know what that space is. The space of features is described by means of a grammar where the language accepted by the grammar is some subset of an existing programming language. Each sentence from the grammar will be an expression which can compute the value of a feature when run over the compiler's internal representation of the current code section. The compiler writer must choose what space of features to search over and design a feature grammar to represent that space. The design process itself is not automated; the compiler writer may make features which explore the abstract syntax tree, control flow graphs and any other data structures available in the compiler. Feature grammars and the compiler's language grammar need not be related.

Figure 5.4 shows a simple grammar describing a set of such features in a pseudo-code style. These example features can be computed on expressions from the toy language and, in this case, each feature counts the number of sub-trees in the expression which match a pattern. Figure 5.5 shows an example feature from this grammar and an evaluation against a sample program fragment, showing the matching sub-trees. Applying this particular feature to this piece of code yields the value three.

The next section describes how a feature will be computed.

5.3.2 Feature Evaluation

The feature grammar defines a set of features, each of which is a sentence from that grammar. Once a feature has been extracted from the grammar it will need to be evaluated against the compiler's internal data to compute the feature values. A grammar is able to produce sentences which are a subset of some particular programming language. In this way an interpreter or compiler for the features already exists in the form of whatever compilers or interpreters there are for the underlying language or if one does not exist.

To give a concrete example, figure 5.6 shows how the grammar from figure 5.4 (which was only presented in pseudo-code) might be implemented in C. Every sen-

tence from the grammar is a valid C program which can be compiled with any standard C compiler. When run on an AST data file of a program from the toy language, the feature program will print out the computed value of its feature.

Features can be arbitrarily complicated, since typically the underlying language is Turing equivalent as long as the desired features can be expressed through the context free grammar. The next section deals with how to handle cases were the restriction to be context free is too limiting.

5.3.3 Semantic Actions

Context free grammars are subject to a number of limitations which restrict the types of sentences that can be created. For example, consider the feature in figure 5.7, which can be computed over arrays of integers. Suppose that the intention is to have features which allow deep nests of such for loops with a computing expression in the body of the innermost loop; the number of variables available to the innermost expression increases with every containing loop. This cannot be expressed with a CFG because a CFG cannot convey the semantics involved.

The situation has parallels in the world of program parsing. There, a CFG is used to describe the syntax of a language and the semantics of the language are embedded as 'semantic actions' (Aho et al. (1986)) in the grammar. In parsing, these actions allow arbitrary code to be run during the parsing process; for example, symbol tables are updated and checked.

The system has a similar mechanism, allowing the grammars to produce more complicated features. Semantic actions are embedded in the grammar and whenever a production is selected the actions are run in the appropriate place. These semantic actions are snippets of arbitrary code which can update state and print values. For example, figure 5.8 shows a grammar which generalises the feature from figure 5.7. The first action in line 1 initialises a depth counter to zero; this action is executed before any rules are expanded. The next action is in line 7 which prints the current variable name and increments the depth. The final action, in line 13 prints a random variable name from those available.

Semantic actions can be placed on entry to or exit from the whole grammar, rules and productions, as well as inside productions, as shown by the example. These additional capabilities need only be used sparingly but allow extremely detailed and powerful control over the features that can be produced.

```
1  <feature>  ::=
2        "#include <stdio.h>"
3        "enum Type {CONSTANT, VARIABLE, PLUS, TIMES};"
4        "struct Node {int type; Node* left; Node* right};"
5        "bool match(Node* ast, Node* pattern) {"
6        "   if(pattern == null) return true;"
7        "   if(pattern->type != ast->type) return false;"
8        "   return match(ast->left, pattern->left) && "
9        "          match(ast->right, pattern->right);"
10       "}"
11       "int countNodesMatching(Node* ast, Node* matcher) {...}"
12       "Node* readAST(char* filename) {...}"
13       "int main(int argc, char* argv[]) {"
14       "   Node* ast = readAST(argv[1]);"
15       "   Node* pattern = " <matches> ";"
16       "   printf('%d\n', countNodesMatching(ast,pattern));"
17       "   return 0;"
18       "}"
19 <matches> ::= "new Node(CONSTANT, null, null)"
20           | "new Node(VARIABLE, null, null)"
21           | "null"
22           |  "new Node(" ( "PLUS" | "TIMES" ) ","
23                        <matches> "," <matches> ")"
```

Figure 5.6: A version of the grammar from figure 5.4 in C. A few details have been elided for clarity. Any sentence from the grammar is a C program which can be compiled and run. When given an AST from the toy language, the C program will print the computed feature value.

Each program from the grammar first reads in the AST of the toy language as a tree of `Node` elements into variable `ast` (line 14). In line 15 a tree is created that will be the pattern to match against; the grammar to create the pattern is in lines 19-23. Line 16 prints the number of nodes in `ast` that match the `pattern`; the `countNodesMatching` function will use `match` function (lines 5-10) as a subroutine. The `match` function tests whether a given pattern matches a particular node of the AST.

```
int feature(int A[]) {
    int s = 0;
    for(int i0 : A) {
        for(int i1 : A) {
            s += i0 + i1
        }
    }
    return s;
}
```

Figure 5.7: A feature over arrays, written in Java. If the feature were generalised so that the loop nest could be of increasing depth, then this would not be representable as a CFG. CFG's do not support these semantics for sentence generation any more than they do for sentence parsing.

```
{int depth = 0;}
<feature> ::= "int feature(int A[]) {"
              "    int s = 0;"
              <nest>
              "    return s;"
              "}"
<nest>    ::= "for(int " {print("i" + depth); depth++} ": A) {"
              <nest>
              "}"
            | "s += " <expr>
<expr>    ::= <expr> " + " <expr>
            | <var>
<var>     ::= {print("i" + random(0 to depth-1))}
```

Figure 5.8: A feature grammar with semantic actions, generalising the feature from figure 5.7. Semantic actions are found between braces.

```
1 <feature> ::= "countNodesMatching(" <matches> ")"
2 <matches> ::=[weight=10] "isConstant"
3             |[weight=10] "isVariable"
4             | "isAnyType"
5             |  ("isPlus" | "isTimes")
6                ("&& leftChildMatches(" <matches> ")")?
7                ("&& rightChildMatches(" <matches> ")")?
```

Figure 5.9: A small modification to the grammar of figure 5.4, showing weighted productions. In this example, the researcher has decided that features testing for constants or variables should be ten times more likely than others.

The next section describes how the grammar can be annotated to change the bias for different areas of the space; i.e. how the researcher may set his expectations about the value of different features.

5.3.4 Production Weighting

There are times when the researcher believes that some part of the feature space is more likely to be useful than others. He would like to be able to influence the grammar to reflect his interest in different portions of the feature space. The grammar definition language allows productions to be weighted, changing the relative probabilities of productions. This makes the grammars probabilistic context free grammars (pCFGs) and allows the researcher to express their interest in some features over others.

Figure 5.9 shows a simple example where some productions are annotated with weights. Now, when choosing between productions for the `<matches>` rule, the first two will be ten times more likely than before. The researcher has changed the probability that some parts of the space will be explored compared to others.

Another example is shown in figure 5.10. The two grammars in that figure both describe the set of decimal digits, `"0"` to `"9"`. However, because of their construction they have very different preferences for different digits. In the first, figure 5.10(a), since the productions are chosen with a uniform probability, the digits will appear with equal likelihood. In figure 5.10(b), however, the characters `"8"` and `"9` are equally likely, but `"7"` is twice as likely as those, `"6"` is twice as likely as `"7"` and so on until `"0"`, which will be chosen with probability 0.5 is 256 times more likely to be chosen than `"8"` or `"9"`.

```
1 <digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

(a)

```
1 <digit> ::= "0" | <d1>
2 <d1>    ::= "1" | <d2>
3 <d2>    ::= "2" | <d3>
4 <d3>    ::= "3" | <d4>
5 <d4>    ::= "4" | <d5>
6 <d5>    ::= "5" | <d6>
7 <d6>    ::= "6" | <d7>
8 <d7>    ::= "7" | <d8>
9 <d8>    ::= "8" | "9"
```

(b)

Figure 5.10: Two different grammars for choosing a decimal digit. Both grammars recognise exactly the same language but they have significantly different preferences.

The grammars of figure 5.10 demonstrate that the precise form of a grammar can have a dramatic effect on preferences for different parts of the space. Sometimes the most natural way of defining the grammar would lead to an unacceptable bias for some features over others. It may be quite difficult to alter the structure of the grammar to even out these issues, and instead weighting can be used to adjust the grammar back into what the researcher was looking for. In figure 5.11, the grammar from figure 5.10(b) has been weighted to have the same probabilities as the grammar in figure 5.10(a).

There are occasions where semantic actions are useful is in setting production weights. The weights can, in fact, be calculated from arbitrary code. If, for example, it is desired that the loop nest from figure 5.8 should become more likely to terminate the deeper it becomes and yet not to be deeper than 5, then the following weights could have been put on the productions at lines 7 and 10 as shown in figure 5.12.

5.4 Generating Features from Grammars

This section discusses how features can be randomly created from a grammar and the design challenges that creates. The main challenge involves ensuring that sentence creation finishes because it is quite possible to construct infinitely recursive grammars.

```
1  <digit> ::= "0" |[weight=9] <d1>
2  <d1>    ::= "1" |[weight=8] <d2>
3  <d2>    ::= "2" |[weight=7] <d3>
4  <d3>    ::= "3" |[weight=6] <d4>
5  <d4>    ::= "4" |[weight=5] <d5>
6  <d5>    ::= "5" |[weight=4] <d6>
7  <d6>    ::= "6" |[weight=3] <d7>
8  <d7>    ::= "7" |[weight=2] <d8>
9  <d8>    ::= "8" | "9"
```

Figure 5.11: A weighted grammar, similar in structure to that of figure 5.10(b) but which has the equal probability of choosing each digit just as in figure 5.10(a).

```
7  <nest>    ::=[weight=depth>5 ? 0 : 1] "for(int " {print...}
10             |[weight=depth * 2]  "s += " <expr>
```

Figure 5.12: Modifications to the grammar of figure 5.8 to make the loop nest favour shallower nests and to have a hard depth limit of five.

How to expand a feature from the grammar is described in section 5.4.1. The causes of infinitely recursive grammars are covered in 5.4.2. The way to solve the problems of infinite recursion is discussed in 5.4.3 and then the consequences of that solution are talked about in section 5.4.4.

5.4.1 Feature Expansion

Now that there is a grammar describing the space of features, any number of features can be generated from it. One need merely start at the root rule of the grammar (which, in the case of the grammar in figure 5.4, is rule `<feature>`) and expand any non-terminals in it. Whenever there is a choice of production to expand they are chosen from randomly using roulette wheel selection (Back, 1996) where the probability of choosing each production is proportional to its weight. By continuing until there are no more non-terminals left in the sentence there will be a finished feature.

For the example in figure 5.5, a derivation is given in figure 5.13. Step one starts with the root rule of the feature grammar - placing a single non-terminal as the current sentence. In step two the non-terminal is replaced by the only possible rule, leaving still only one non-terminal to be replaced. In step three the `<matches>` non-terminal is

```
1. <feature>
2. "count-nodes-matching(" <matches> ")"
3. "count-nodes-matching(
   is-times &&
   left-child-matches(" <matches> ") &&
   right-child-matches(" <matches> "))"
4. "count-nodes-matching(
   is-times &&
   left-child-matches(is-plus) &&
   right-child-matches(is-constant))"
```

Figure 5.13: Derivation of the example feature from figure 5.5.

```
<A> ::= <A> "a"
```

Figure 5.14: Infinitely recursive grammar. The only sentence the grammar recognises is an infinite sequence of a's. Attempting to generate a sentence from this grammar will never finish because there will always be a non-terminal <A> left un-replaced.

replaced; there are five productions and the last is randomly selected; the non-terminal is replaced with value of the production giving two non-terminals to replace. Finally, in step four the remaining <matches> non-terminals are replaced.

5.4.2 Problems of Recursion

Whilst ambiguities cause problems in parsing sentences from grammars, sentence production suffers from infinite recursion. Perhaps the simplest example of this is in the grammar in figure 5.14 which obviously produces an unending string of '*a*'s. Attempting to generate a sentence from this grammar will not succeed. The grammar definition language allows embedded actions, similar to semantic actions in parser generators, which allow such problems to be manually broken - by, for example, imposing a depth limit on the recursion. However, in practice such grammars are unlikely to be seen.

More subtle recursion issues are caused probabilistically. Consider the grammar in figure 5.15. The language it recognises consists of odd numbers of consecutive '*a*'s. However, if the two productions are chosen from uniformly at random, then it is likely to get very long strings. The probability that a string of n non-terminals, $AAA...A$, will contain fewer non-terminals after each is expanded once is given by

```
<A> ::= <A><A><A> | "a"
```

Figure 5.15: Probabilistically recursive grammar. At any point in the expansion of the sentence it is possible for the all non-terminals to be replaced with terminals. However, since the two productions are chosen with equal probability and the first production generates so many recursive non-terminals, there will most likely be an explosion of non-terminals and the sentence will become longer and longer.

$I_{1/2}(2n/3, n/3+1)$, where I is the regularised incomplete beta function. As strings contain more non terminals they become increasingly likely to grow at each expansion. Production weighting solves these issues, as described next in section 5.4.3.

5.4.3 Avoiding Runaway Sentence Expansion with Production Weights

Explosive sentence lengths can be avoided by changing the probabilities of different productions. If, in the example from listing 5.15, the weight of the first production had been less than one third of the weight of the second production, then the expected length of a sentence after replacing each non-terminal once would be less than the original; the sentence expansion would not explode.

Deciding the appropriate weights for productions is not generally possible to do analytically. This is due to the presence of semantic actions (see section 5.3.3) which introduce arbitrarily complex code into the grammar expander. However, as can be seen from figure 5.16, once weightings create a non-explosive grammar there is relatively small sensitivity to the weight values. Thus, it is quite easy to be conservative with weightings, the grammar will not suffer much for it; trial and error produces acceptable results with very little time or effort.

5.4.4 Short Sentence Bias

As described in the previous section, the grammar must weight productions to prevent runaway, explosive sentences. A consequence of this is that the grammar system is biased towards short sentences. Figure 5.16 shows the sentence length bias for the grammar in listing 5.15. It can be seen that short sentences are produced, even when the productions are weighted close to the threshold at which run away expansion begins. For this grammar, the vast majority of sentences will be shorter than ten characters long.

Creating grammars that give rise to mostly short sentences has some drawbacks. In essence, by avoiding the grammar exploding, the preference for short sentences prevents the exploration of much of the grammar. Often, after generating a few thousand random sentences, the system typically recreates sentences that have already been seen; exploration effectively stops.

Overcoming the bias toward short sentences is a side effect of searching the feature space, which will come be covered in the next section 5.5.

5.5 Support for Searching the Feature Space

The simplest way to search the feature space is to randomly generate features as described in section 5.4. Thousands of features can be created in a matter of seconds which can then be evaluated. The bias towards short sentences, however, (see section 5.4.4) means that this approach will be practically limited to a small portion of the complete feature language accepted by the pCFG. Early experiments found that the amount of the feature space that could be reached was much smaller than expected.

One might try to use semantic actions to alter the bias of the feature space as the search begins to saturate its exploration of short features. This, however, is difficult to arrange, placing a heavy burden on the writer of the grammar to add large amounts of fragile code that are not directly concerned with describing the feature space. One could also arrange the weights of productions to favour long sentences. This quickly leads to runaway sentences in the feature generator. Even if the generator is rigged to bail out when a sentence becomes too large, the generator then spends most of time creating features that fail or that have already been seen before.

Fortunately, the problem of short sentence bias is solved as a side effect of different search techniques than the naïve random approach. Suppose there is some feature to start with, chosen from the biased space. Small modifications can be made to it; possibly shortening it, maybe lengthening it. The feature will no longer be bound by the imposed bias of the pCFG (the bias will now only inform where the search should start and its direction, not limit the scope of that search).

The next section, 5.5.1, describes the trees that are searched over. Discussed, in section 5.5.2 is how the trees are repaired after modification, which is required for search operators that modify the trees and are themselves introduced in section 5.5.3. Finally, section 5.5.4 discusses the differences between our system and other, older systems.

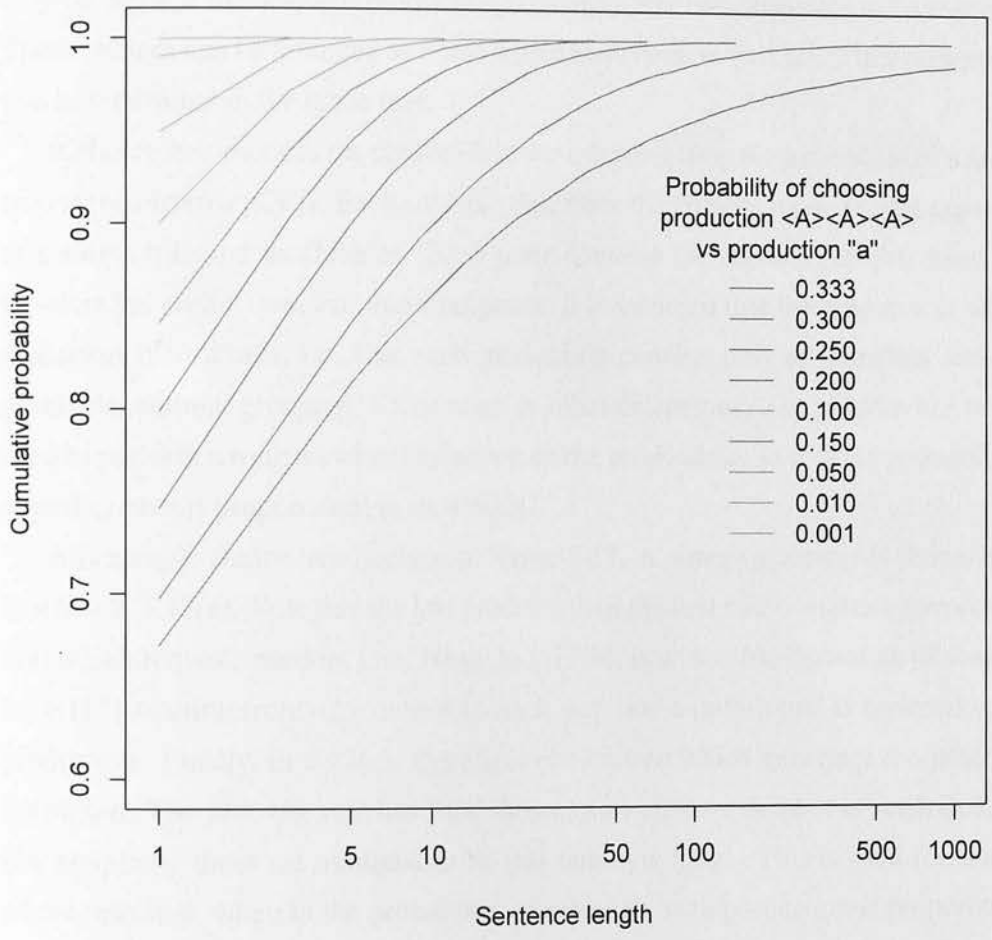


Figure 5.16: Cumulative probability of getting a sentence of a given length when expanding grammar `<A> ::= <A><A><A> | "a"` with different weights for the two productions. Even as the weights approach the beginning of runaway sentence generation (which first happens when production `<A><A><A>` is one third as likely as production `"a"`) there is significant bias towards short sentences.

The graphs also help to explain why setting weights by trial and error is so easy. Being conservative with the weights does not have a large effect on the sentence length; they will always be short for feasible weights.

5.5.1 Choice Trees

The entities that are searched over in the system are the trees of choices that are made during the construction of a sentence or feature. For each rule there are some number of productions that must be chosen between and depending on that choice, further choices may be made if the selected production contains other non-terminals to be expanded. These choices can be arranged as a tree where each node in the choice tree corresponds to a non-terminal in the parse tree.

A choice tree encodes the choices that were made during the generation of a feature or sentence from a pCFG. Each sub-tree describes the choices made for the expansion of a single rule and its children. Each node contains the random bits that were used to select the production. For these purposes, it is assumed that the grammar is written in Backus Naur Form; i.e. that each production consists only of terminals and non-terminals, without grouping, Klene stars or other extensions. The random bits will be used to perform a roulette wheel selection of the productions so that the probability of selecting each is proportional to its weight.

An example choice tree is given in figure 5.17. A simple grammar is shown in the first block, 5.17(a). Note that the last production of the first rule contains a semantic action which requests random bits. Next, in 5.17(b), is a possible derivation of sentence `babR(10)` starting from `<A>`, where in each step one non-terminal is replaced with a production. Finally, in 5.17(c), there is a choice tree which generates the preceding derivation. The first `<A>` rule has four choices so a random number is generated, `220` (for simplicity these are assumed to be just one byte long). This is used for roulette wheel selection, wherein the probability of selecting each production is proportionate to its weight. There are four choices for `<A>`'s productions and since all productions have unit weights roulette selection is equivalent to the modulo function. The random number, `220`, modulo 4 is 0, so the first production, `<A>` $\rightarrow$ `<A><A>`, is chosen. Subsequent nodes in the tree represent the remaining replacements of non-terminals. In the cases where the `<B>` non-terminals are replaced there is no choice; denoted by `X`. Note that the last leaf node corresponds to the production with a semantic action, so the node has one set of random bits for the production choice and one set as used by the semantic action. The choice tree itself consists only of the nodes and their random bits.

Choice trees can be used in two complimentary fashions; one records the choices made during the expansion of a feature and once recorded, it can be used to replay

those choices to recreate the exact same feature as before. During the recording any random bits are remembered and sub-trees delimited according to the rules and productions of the grammar. When replaying, the source of random bits is replaced by those previously recorded.

It should be noted that semantic actions (see section 5.3.3) can make use of random bits. This means that additional bits may be required when a particular production is chosen or during the evaluation of a production's weight. These bits are simply appended to the random bits used to choose the production during recording and are delivered up during replay.

5.5.2 Repairing choice trees

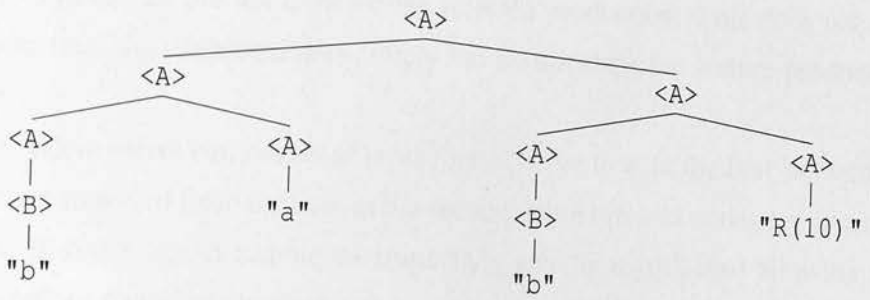
The search operators in section 5.5.3 perform modifications to the choice trees. They may change a choice tree so that there are insufficient random bits to complete the replaying of the tree to create a feature. For example, a mutation search operator may delete some sub-tree or change a simple production without children to one which requires several children. This section describes how these trees are repaired so that they are always valid; no search operation, no matter how drastic can create an invalid tree.

The mechanism for repairing trees is that whenever additional bits are needed during playback, they are created at random and recorded back into the choice tree. During playback, whatever bits are present in the tree in the correct places are made use of. However, if at any point there should be missing information, that will cause a change from playback mode to recording mode until the required information is made up. In this way, reading a choice tree can alter the tree as a side effect, but at no point is the tree starved of information.

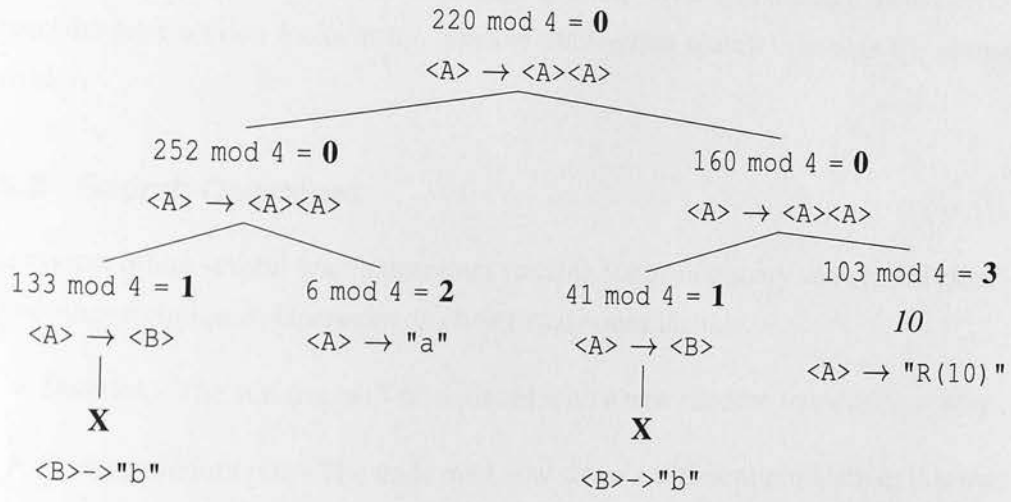
Figure 5.18(a) shows the choice tree from figure 5.17 that has been mutated during a genetic search so that the random bits of one node have changed (marked with a box). The system begins to replay the tree (b), expanding the root non-terminal making choices according to the data in the tree. This proceeds just as normal until the mutated node is encountered. At this point the random bits in the selected node choose a different production from the one in the original tree; the `<A>` becomes `<A><A>` not `"a"`. Now, if the tree were complete there should be two child nodes beneath the mutated node, which are missing. To solve this issue the system begins to randomly create any nodes it needs, building a new sub-tree, rooted at the mutated node (c). Once returned

```
1   <A> ::= <A><A>                                  0
2         | <B>                                     1
3         | "a"                                     2
4         | {print("R("+randomInteger+")")}          3
5   <B> ::= "b"
```

(a)



(b)



(c)

Figure 5.17: An example choice tree. A simple grammar is shown in (a) where the productions of the rule for non-terminal <A> are numbered for clarity. A possible derivation of sentence babR(10) is given in (b). (c), is a choice tree for the derivation in (b); when selecting a production for rule <A>, a random number is needed (in this example, from $[0, 255]$), the production is then chosen by roulette wheel selection (equivalent to mod 4 in this case) to give the production number. Not all rules offer choices and so do not need random bits (marked with **X**). Production 3 of rule <A> requires additional random bits, in this case, 10.

from repairing the mutated node, the system begins replaying, just as normal, yielding a complete feature and updating the tree so that it is now complete and has remembered the new information that was used to repair the tree.

It may also happen that modifications to a choice tree increase the information in the tree, beyond what is needed. This occurs if bits that would select a node with many children are changed so that the node will only have one child, or additional bits are added to a node that are not used for the selected production. This does not damage the choice tree, the additional data simply has no effect on the feature produced from the tree.

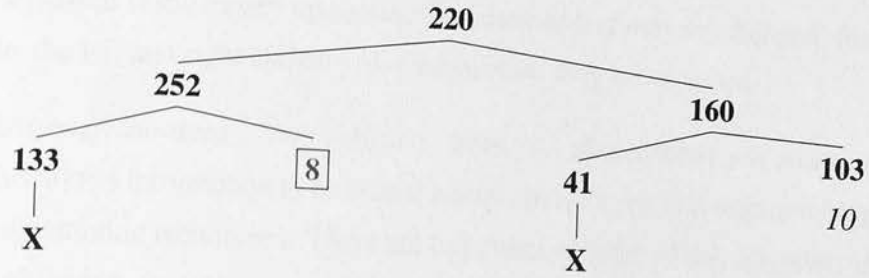
The system offers two modes of replaying a choice tree; in the first any extraneous data are not removed from the tree, in the second, extra bits and nodes are pruned from the tree. In some search techniques (especially genetic algorithms) allowing genetic information to contain redundant information is considered good practice since it is believed to mirror the biological counterparts. Indeed, in genetics, redundant data are given their own term, introns. Both capabilities are provided in the system.

Now that it has been seen that no change to choice trees can damage them beyond repair, the next section looks at the types of choice tree search operators the system provides.

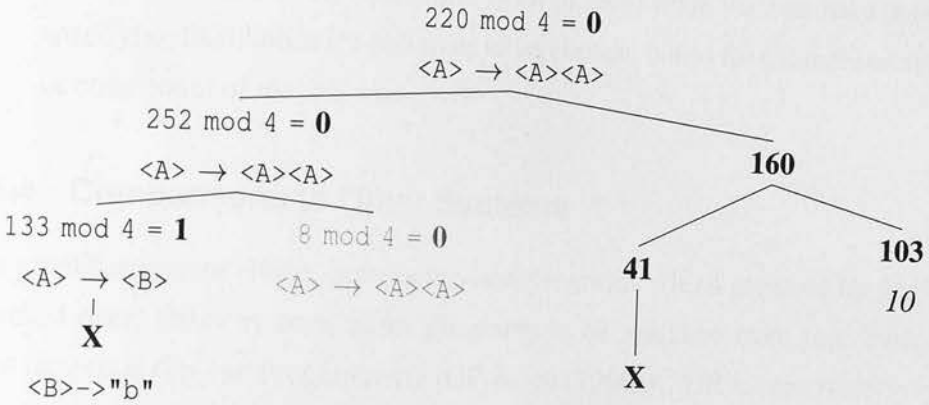
5.5.3 Search Operators

The system offers several search operators suitable for evolutionary search, hill climbing or other techniques. Operators on choice tree nodes include:

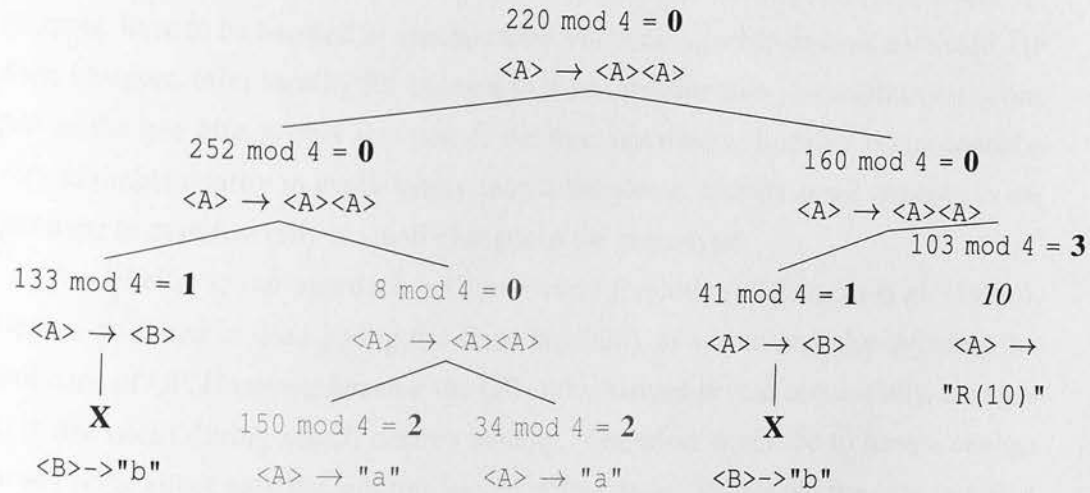
- Deletion - The sub-tree will be replaced with a new random tree during replay.
- Change random bits - The node may now select a different production; this may cause some children to be missing or irrelevant. Often rules have many productions with similar forms (e.g. a rule for binary expressions may have many productions with the same two non-terminals); changing random bits, might simply swap one such production for another. Constants are also generated from random bits, so this type of mutation can search through different constants.
- Shuffle children (several variants) - If the children are from unrelated non-terminals then this might be equivalent to deleting all the children and recreating them (although with some pre-specified random bits). If children are related, such as



(a)



(b)



(c)

Figure 5.18: Repairing a choice tree. In (a) the choice tree from figure 5.17 has one node mutated (marked with a box). Replay begins reading the choice tree in (b) as normal until the mutated node is reached when a different production is chosen. The new production needs more data than the tree contains, so that is generated and recorded in the tree (c). There after the remains of the tree are unchanged.

children of many binary operators, then their orders may be changed; for example, the left and right children of a subtraction may be swapped.

- Crossover sub-trees - Two sub-trees from two choice trees are swapped over. This allows information to be shared across choice trees as is required for genetic programming techniques. There are two main variants of this operator: the first is oblivious, it chooses any sub-trees from the two choice trees; the second is more targeted - it first expands both trees, remembering which the names of the rules that each node in the trees relates to. Then it prefers to select two sub-trees which relate to the same rule. Both of these main variants have parameters specifying likelihoods for sub-trees to be chosen, based for example on the depth or node count of the sub-tree.

5.5.4 Comparisons to Other Systems

The grammar system allows expressions and programs which are used for features to searched over. Other systems allow programs to be searched over, too. Perhaps the most famous is Genetic Programming (GP, Koza (1990a)). GP, as described in chapter 3, builds an expression tree where each node must have the same return type. This is very restrictive, leading to difficulties when there are different types in the system, like booleans, floats and integers. GP also suffers from serious problems extending to anything other than expression trees; the definitions of functions and loops, for example, have to be handled as special cases which can quickly become unwieldy. GP does, however, offer locality for changes to the expression trees; a modification in one part of the tree affects only that part of the tree, not others. Locality is considered a very desirable quality in evolutionary search because it permits small changes to the genotype to manifest only in small changes to the phenotype.

Most similar to our approach is Grammatical Evolution (GE, Ryan et al. (1998)). Here, a grammar is used giving the same flexibility as we do and also avoiding the problems of GP. However, because the GE codon stream is read sequentially, changes to it that occur during search destroy locality. The ideal would be to have a change in any node affect only the sub-tree rooted at that node. This is not the case in GE, a change early in the codon stream can make every other node subsequently generated be completely different. This causes problems for GE searches since the majority of mutations cause massive damage and the search degenerates into a random search (which is not the case with GP). Our system, based on choice trees suffers no such problems.

Any change to a sub-tree in the choice tree modifies only the corresponding sub-tree in the resulting parse tree. In this way, our system combines the good searchability of GP with the expressive power of GE.

5.6 Summary

This chapter has shown how manually writing features is error prone and difficult and how the space of possible features is infinite. A method of describing families of features by using grammars was introduced and numerous associated problems explored and resolved. Finally, the elements essential to searching the feature space were described. The next chapter explains how to search the feature space in practice and details an experiment that demonstrates the effectiveness of the system.

Chapter 6

Searching for Features

The last chapter explained how families of features can be built from grammatical descriptions. In this chapter we search the space of features to find good features which most improve the machine learning algorithm's performance. Although prior works have searched over the model space (Stephenson and Amarasinghe (2005)), this is the first to generate features, searching over the feature space.

We evaluated our technique on an extensively studied problem: loop unrolling. Loop unrolling is an optimisation performed by practically every modern compiler and we study its effect in a widely used open-source compiler, GCC. Furthermore, machine learning has been successfully applied to loop unrolling (Stephenson and Amarasinghe, 2005), allowing direct comparison. Given the mature nature of this problem, it should be a challenging task for a new technique to show additional improvement.

The remainder of this chapter is organised as follows. Section 6.1 presents an overview of the system explaining the major components and how the feature grammars from the last chapter are used within it. Then in section 6.2 the particular feature grammar used for the GCC experiments is laid out, followed in section 6.3 by a demonstration of the benefits of generating features. Our experimental setup, methodology and results are presented in sections 6.4, 6.5, and 6.6 respectively. This is followed by some concluding remarks in section 6.7.

The main contribution of this chapter is an holistic, automated feature generation system and its application to loop unrolling in GCC.

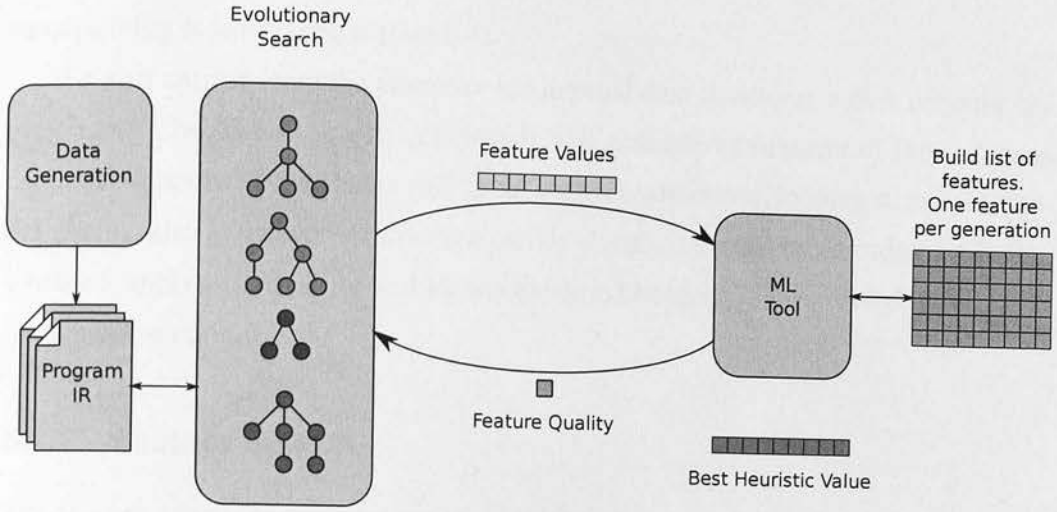


Figure 6.1: Overview of the system.

6.1 Overview

This section presents a high-level overview of the system, illustrated in figure 6.1. The system is comprised of the following components: *training data generation, feature search* and *machine learning*. The training data generation process extracts the compiler's intermediate representation of the program (as described in section 6.2) together with the optimal values for the heuristic we wish to learn. Once these data have been generated, the feature search component explores features over the compiler's intermediate representation (IR) and provides the corresponding feature values to the machine learning sub-system. The machine learning sub-system computes how good the feature is at predicting the best heuristic value in combination with the other features in the base feature set (which is initially empty). The search component finds the best such feature and, once it can no longer improve upon it, adds that feature to the base feature set and repeats. In this way, we build up a gradually improving set of features.

6.1.1 Data Generation

In a similar way to existing machine learning techniques, we must gather a number of examples of inputs to the heuristic and find out what the optimal answer should be for those examples. Each program is compiled in different ways, each with a different heuristic value. We time the execution of the compiled programs to find out which heuristic value is best for each program. Due to the intrinsic variability of the execution times on the target architecture, we run each compiled program several times to reduce

susceptibility to noise (see section 6.4).

We also extract from the compiler the internal data structures which describe the programs. The feature search component will generate summaries of these data as candidate features. Typical data will be the abstract syntax tree, looping structures, use-def chains, etc. Whatever information can be extracted should be recorded, including whatever analyses are performed by any existing heuristics. This process is described in detail in section 6.2.

6.1.2 Feature Search

The feature search component maintains a population of feature expressions, represented as choice trees (described in section 5.5.1). The expressions come from a family described by a grammar derived automatically from the compiler's IR. Evaluating a feature on a program generates a single real number; the collection of those numbers over all programs forms a vector of feature values which are later used by the machine learning sub-system. The construction of grammars and their uses are discussed in section 5.3.

The search component uses a evolutionary search over choice trees with the search operators described in section 5.5.3. These allow genetic mutations and matings of the choice trees. A population of choice trees is kept and sorted according to a fitness function. After the trees are ranked, a new population of the same size is created. Each member of the new population is created by mutations and matings or, rarely, by just simple copying of the members of the previous population. The selection process to determine which individuals will participate in creating the next generation is tournament selection. In tournament selection a small number of individuals are picked uniformly at random and, from these, one is chosen such that the probability of it being the best is highest and the probability for each below it in the ranking is exponentially lower. This ensures that fitter individuals are more likely to contribute their genetic material to the new generation.

The fitness function, which allows the search component to compare the quality of any two choice trees, is the speed-up that would be obtained by using a machine learning model in place of GCC's unrolling heuristic. The model is built from the new feature and the previously fixed set of features, as described in the next section, 6.1.3.

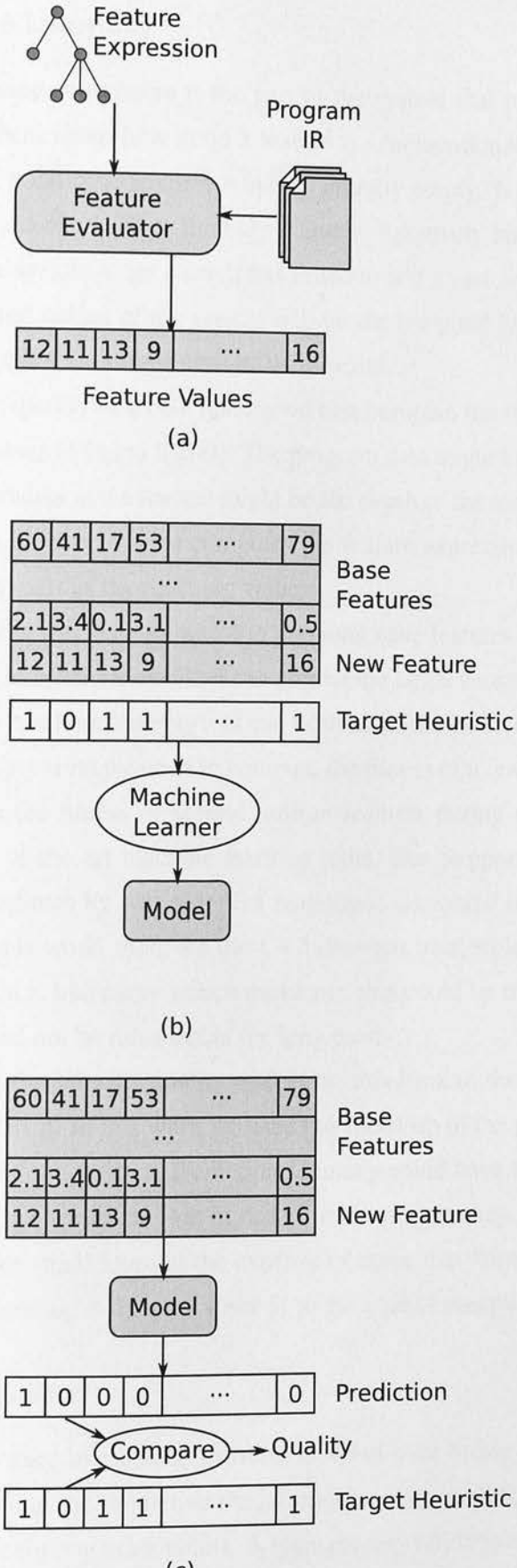


Figure 6.2: Machine learning used to determine the quality of a feature. Each column of the vectors corresponds to the feature values, prediction and target heuristic of a single training benchmark. The target heuristic is determined by iterative compilation.

6.1.3 Machine Learning

The machine learning sub-system is the part of the system that provides feedback to the search component about how good a feature is. As mentioned above, the system maintains a list of good base features which is initially empty. It repeatedly searches for the best next feature to add to the base features, iteratively building up the list of good features. The system stops when it has failed to add a new feature that improves the results. The final output of the system will be the list good features at the end of the search. Figure 6.2 shows the details of the process.

To evaluate the quality of a new feature we first compute the feature values across all programs (as shown in figure 6.2(a)). The program data might be the IR for loops of a number of benchmarks and a feature might be *the depth of the loop times the number of basic-blocks*. A runtime system computes the feature expression on each program datum, yielding a vector of the resulting values.

We then combine this feature with the previous base features and ask a machine learning algorithm to learn a model that can predict the target heuristic value (shown in figure 6.2(b)). Any machine learning tool can be used, however, since the tool will have to learn a model every time we need to compute the fitness of a feature it must be fast; we might compute the fitness of several million features during our search process. Some of the state of the art machine learning tools, like Support Vector Machines, are very slow, sometimes by two orders of magnitude compared to simpler tools like decision trees. In this work, then, we use C4.5 decision trees which are considerably faster. However, since, like many search problems, this could be trivially parallelised, the slower tools need not be ruled out in the long term.

Finally, we test the model's quality and report this back to the search component (shown in figure 6.2(c)). In this work we used the speed-up of the prediction to be the fitness of the model for a feature. Prediction accuracy could have been used, amongst other possibilities, but we found that in this case the system may concentrate on improving accuracy for small loops at the expense of those that dominate the execution time. Improving speed-up is a much closer fit to the goal of compiler optimisation.

6.1.3.1 Cross Validation

Cross validation is used in machine learning to avoid over fitting to the training set. Over fitting means that the predictive model may be very good for the points in the training set but poor for any other points. A typical cross validation scenario partitions

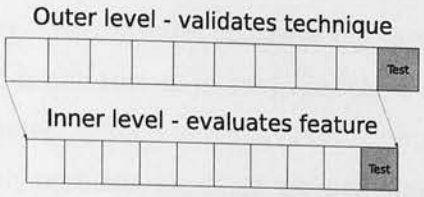


Figure 6.3: Two level cross-validation.
The two levels of cross validation are shown, with one test set held out in each. The outer level is used to evaluate the technique and the inner one is used to evaluate a feature.

the input data into a number of sets (say ten). One set is kept out and called the *test* set, the remainder (typically $\frac{9}{10}^{\text{ths}}$ of all the data) is called the *training* set. A model is trained on the training set and it predicts values for the test set. This is repeated for each partition so that each partition is the test set once and predictions are created for all of the input data. The total prediction (albeit composed from different models) is then evaluated for quality.

We have two levels of cross validation. The outer level is the more typical and is used to determine how well the feature generator works. This level means that the program data is partitioned, providing a training set to the whole feature search, as in figure 6.2(a-c), and complete sets of features are created for each partition. The models learned from those sets are used to predict values for the test sets and the whole prediction is used to evaluate how good the method is. This is no different from most machine learning experiments.

The inner level, on the other hand, is used to prevent the feature generator over fitting while it is searching for features. The step shown in figure 6.2(b) is actually cross validated (we use ten-fold cross validation, so ten models are created for the ten training set/test set pairs). The multiple models combined, build the prediction in figure 6.2(c) over the several test sets.

At no point will the inner level see the outer level's test set. The inner level's training and test sets come from the outer level's training set only. Figure 6.3 shows how the two level cross-validation is done.

6.1.3.2 Parsimony

In practice, the system uses additional information to the quality metric described above. It happens that the feature expressions learned by evolutionary search can

quickly become very long. Two features can have the same results but have different lengths (for example, if one feature is `loop.depth`, a more complicated feature with no more predictive power is `loop.depth+(1+2)loop.depth`.

In order to address this problem, we adopt the well known genetic programming methodology of rewarding parsimony. If the objective function computed by the machine learning tool for two features gives the same value then we determine that whichever feature expression is shorter is the better.

6.2 Grammar for Loops in GCC

This section shows one of the grammars created. This is a grammar for generating features over loops at the register transfer language of GCC (RTL) and is used in the experiments of this chapter. The grammar contains many tens of thousands of productions because of the large number of different data-types in GCC's internal representations. The grammar is automatically created by observing the types of data that GCC uses internally; doing this is essential because of the grammar's size but also prevents the grammar needing to be hard coded, allowing modest changes GCC without requiring an engineer to rewrite the grammar.

An overview of the system is shown in figure 6.4. The system builds three main outputs from observing GCC's internal representation of the benchmarks; the first is a suite of Java classes representing the grammar, the second is a custom feature evaluation language and finally, the third is a compact and efficiently searchable version of the benchmark data. Subsequent sections describe the major components of the grammar generator.

6.2.1 Data Generation

The first stage of the system extracts GCC's internal data structures for each of the benchmarks into XML files. *libPlugin* (see chapter 4) provides a plug-in to perform this data extraction.

In RTL, instructions are in an algebraic form with a treed, list-of-lists representation. Each node in the RTL may have some number of attributes. The RTL representation of the loops is extracted, augmented to include the structure of the basic blocks in the loop and the RTL instructions contained within their blocks. Also exported is any information GCC can compute at that time, such as estimated block frequencies, loop

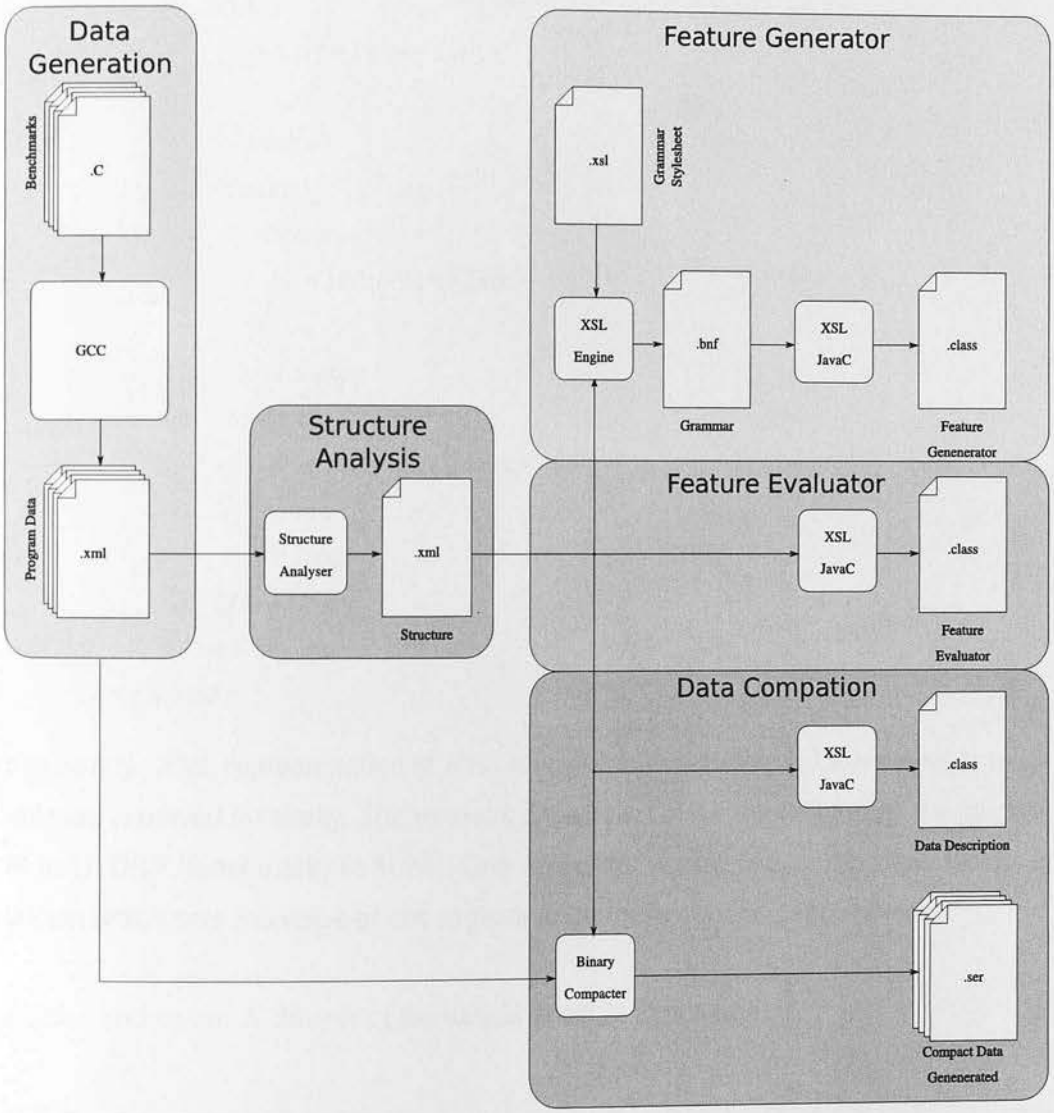


Figure 6.4: Overview of the grammar creation system for RTL loops in GCC.

```
<loop name="UTDSP.fft_1024.fft.3" ... insns="28" expected-iter="11">
  <basic-block index="10" ... frequency="9100" loop-depth="3">
    ...
    <insn ...>
      ...
      <set ...>
        <reg ... mode="SF">
          <int>112</int>
          ...
        </reg>
        <mult ... mode="SF">
          <reg mode="SF">
            <int>94</int>
            ...
          </reg>
          <reg mode="SF">
            <int>84</int>
            ...
          </reg>
        </mult>
      </set>
    </insn>
```

Figure 6.5: XML representation of RTL. Many of the attributes and values made available are removed for clarity. The example shows part of the third loop from the function fft in UTDSP benchmark, fft 1024. One instruction inside one of the basic blocks is shown which sets the value of one register to the multiplication of two others.

depths, and so on. A flavour of the data is given in figure 6.5.

6.2.2 Structure Analysis

The hierarchical data produced follow a number of relational rules. For example, loops contain basic blocks as children and they in turn contain instructions. Those relationships are never violated. It would be wasteful to create a feature like "*count the number of basic blocks which contain three loops*" since that can never be other than zero.

The grammars constructed are automatically derived from the structural rules of the data to ensure that such impossible features are never generated, improving the

efficiency of the search. Since the rules that GCC's internal data structures follow are not immediately derivable in a machine readable manner from the GCC source code, a simpler approach is taken of examining the XML data files to find the observed structure within.

The XML data files are iterated through and the number of each type of node in the XML is collected; how many children each node type has is recorded; a histogram of child types for each child slot in each node type is built, as is a histogram of attribute values for each node type.

6.2.3 Data Compaction

The XML data produced by *libPlugin* in section 6.2.1 is extremely detailed and because of XML's verbose nature the data files come to several gigabytes in total. Processing these files in their raw form to explore features is very memory intensive and typically causes the machine to thrash. To improve performance, the data files are converted into a compact binary format using the structure document as a guide. The structure document declares all the different types of nodes, their attribute names and values and the children each node can have.

The system first creates a Java class that represents each type of node. Each attribute maps to a field of a suitable type and the node's children are packaged as an array. The structure document is used as source data so that the system knows what Java classes are needed. Next, the XML program data files are read and each is converted to objects of the Java classes and serialised to files. The resulting data is much smaller than the original XML, can be loaded into memory all at once and is much faster to compute feature values over.

6.2.4 Feature Evaluator

Although feature evaluation could be performed by an existing scripting language, generic scripting languages were found to be far too slow for searching over. Compiling features into C programs and Java programs was also tried. In C, the effort of spawning GCC and then the feature program also proved to slow. For Java, the compilation could be performed in process but the resulting classes, after being loaded into memory, tended not be garbage collected, so eventually all memory was exhausted.

Instead, a custom feature interpreter was created. There was no need to create a parser for the interpreter, since the feature grammar can produce ASTs directly, not

just strings. The interpreter supports:

- Getting the types of nodes.
- Getting attribute values of nodes, including determining if attributes are missing.
- Logical, comparison and arithmetic operations.
- Aggregators i.e. summations, finding extrema, etc.
- Iterating over children, descendants.
- Standard control flow operations.
- Pattern matching.

Runtime support for feature evaluation is also computed from the structure document.

The next section describes how features are created that make use of this interpreter.

6.2.5 Feature Generator

This structural information is then used to mechanically create a grammar. Because they are automated and not hard coded, they are easy to update in response to changes in the compiler. The grammars, being machine generated, are quite large; several hundreds of kilobytes long. The transforms used in practice all make sure that trivially impossible features (or rather, uninteresting features which do not relate to possible structures) are never created. They also automatically set production weights to ensure grammars do not suffer from the probabilistic recursion issues described in subsection 5.4.2.

Figure 6.6 shows a pared down snippet of the grammar, giving an example of some of the feature expressions that can be created. The full grammar has many more functional capabilities and is also tuned to the structure of the RTL. This bit of the grammar says how some numerical values can be computed on node from the data: a numeric can be a binary expression of two numerics; it may be the numerical value of an attribute; it may aggregate recursive numerical values of filtered children; it might count the number of children matching come criterion; it might simply delegate to another numeric expression of one of its a children. Matching criteria for selecting child nodes may be the logical combination of other matchers; it may be the result of comparing two numerics; it may check the type of the node; it may test the value of an attribute.

```
<numeric> ::= <numeric> ( "+" | "-" | "*" | "/" ) <numeric>
            | <value-of-an-attribute>
            | ( "sum" | "min" | "max" | "avg" )
              "(for-each-child-that(" <match> "do" <numeric> "))"
            | "count-children-matching(" <match> ")"
            | "on-child" <random> "do" <numeric>
<match>   ::= <match> ( "or" | "and" | "xor" ) <match>
            | "not(" <match> ")"
            | <numeric> ( "<" | ">" ) <numeric>
            | "is-type(" <node-type> ")"
            | <attribute> "=" <value>
```

Figure 6.6: A simplified subset of the automatically generated grammar.

This part of the grammar is replicated many times for each bit of structural data. There are also many other functional concepts encoded in the grammar in similar ways.

6.3 Motivating Example Reprise

In the previous chapter, section 5.2.2 presented an example for which naïve features failed to achieve much of the potential speed-up available, achieving only 14% of the maximum for that example. If, instead, our technique is used to search for the best set of features and train a decision tree over those then the best unroll factor of 11 can be automatically selected, giving the maximum speed-up for the loop in figure 5.1. The path of the decision tree selecting the unroll factor of 11 is also shown in figure 6.7(b) and the features touched are shown in figure 6.7(a). This example shows that while performance of the heuristic can be improved by a machine learning approach, it may ultimately be limited by the features used. By searching the space of features, features better suited to the learning task at hand can be found.

6.4 Experimental Setup

In this section we briefly describe the experimental set-up and how the training data was generated as well as the steps taken to ensure accuracy of measurement.

Name	Value	Feature
f0	6.14 E17	`get-attr(@num-iter)`
f1	308	`count...!is-type(wide-int) ..`
f2	2	`count...is-type(basic-block)...`
f3	5	`max... is-type(basic-block) .`
f4	4	`count... is-type(array_type)`
f5	0	`count... is-type(le) && ...`

(a)

```
1  if( f2 <=4 )
2   if( f5 <= 0 )
3     if( f0 > 8206 )
4       if( f1 > 168 )
5         if( f0  > 6.1E17 )
6           if( f2 <= 3 )
7             if( f1 <= 1247 )
8               if( f4 > 1 )
9                 if( f3 > 4 )
10                  if( f3 <= 6 )
11                    unrollFactor = 11;
```

(b)

Method	Unroll	Cycles	Speedup	% of Max
Baseline	0	406,424	1.0000	0%
Oracle	11	328,352	1.2378	100%
GCC Default	7	418,464	0.9712	-12%
GCC Tree	2	392,655	1.0351	14%
Our Technique	**11**	**328,352**	**1.2378**	**100%**

(c)

Figure 6.7: The path through the learned heuristic (b) for the example in figure 5.1 and the features from our scheme used by that path (a). In (c), the speed-up using our features is compared to that from other methods.

6.4.1 Compiler Setup

To demonstrate the applicability of our approach, we have applied it to loop unrolling within GCC 4.3.1. Loop unrolling is an extensively studied optimisation and there exists prior work (Monsifrot et al., 2002; Stephenson and Amarasinghe, 2005) with which to compare. We extended the compiler to allow unroll factors to be explicitly specified for each loop in a program.

6.4.2 Benchmarks

We took 57 benchmarks from the MediaBench, MiBench and UTDSP benchmark suites. Those benchmarks from the suites which did not compile immediately, without any modification except updating path variables, were excluded.

6.4.3 Platform

These experiments were run on a single unloaded, headless machine; an Intel single core Pentium 6 running at 2.8 GHz with 512 Mb of RAM. All files for the benchmarks were transferred to a 32 Mb RAM disk to reduce IO variability.

6.4.4 Generating Training Data

In order to learn the best unroll factor we need to generate training data where we know the best unroll factor for each of the training loops. To find this we took each loop, one at a time, and unrolled it by different factors, zero to fifteen. This gave a compiled program for which all but one loop has the default unroll factor as determined by GCC's default heuristic. We executed each of these versions of the program a number of times, in each case recording the number of cycles required to execute the function containing the loop that had been altered. We compiled without inlining to increase the independence of loops. In total we gathered data for 2778 loops.

6.4.5 Measurement

One of the difficulties in evaluating the performance of compiler optimisations is the impact of noise on the measured results. For each differently compiled variation of a benchmark we ran that version of the program at least one hundred times. We applied a standard statistical technique to reduce the effects of noise: applying a log transform

and removing outliers outside the 1.5 x IQR (interquartile range). The best unroll factor for each loop was determined as that with the lowest average cycle count (across the 100 runs).

6.5 Experimental Methodology

This section outlines the methodology used when applying the feature search technique to the problem of loop unrolling in GCC.

6.5.1 Searching for Features

The feature generator searches for one feature at a time. It prefers features which, in combination with the features selected by previous steps, most improve the performance of a machine learning tool. The evolutionary search for each feature consisted of a population of one hundred individuals. Each was allowed to run until fifteen generations produced no improvement in the best feature of the population or a maximum of two hundred generations, whichever came first. Search for new features to add was stopped when either two and a half thousand total generations were reached or when we failed to find an improving feature five times.

6.5.2 Cross-validation and Machine Learning

We split the loops into ten groups, keeping one group out for testing so that we can perform ten-fold cross validation. Loops that are used for generating features and later learning a model are *never* used to evaluate the model. Final evaluation is always on *unseen* loops.

The machine learning algorithm used to find the quality of the features was a simple C4.5 decision tree Monsifrot et al. (2002), selected for its speed. When a feature was evaluated, we trained a decision tree on eight of the remaining nine loop partitions, called the training set. We then asked the decision tree to predict the unroll factors for loops in the remaining, ninth part, called the *internal validation* set. This was then used to determine the speedup attained by those unroll factors.

6.5.3 Search, Training and Deployment Cost

It took the system two days to learn the best set of features and model for this problem. Although this is a significant amount of time, it is a one off activity that it is performed "at the factory" and would be easily parallelised. If we consider the amount of time it takes for a compiler writer to develop a good heuristic, this cost is in fact small.

It is theoretically possible for the system to produce extremely computationally expensive features, increasing compile time. The system forces these feature evaluations to time out, giving them at most two seconds to evaluate over all loops (thus, with 2778 loops in 2 seconds, no feature can take more than 0.7ms to compute per loop on average). If a feature times out it is discarded and cannot contribute to the gene pool. We find that the pressure for simpler features means that features rarely time out. The features selected by the system for unrolling have no significant impact on GCC's execution time.

6.6 Results

This section evaluates our technique when applied to loop unrolling, demonstrating that it outperforms existing approaches. We first show the maximum benefit available from loop unrolling across the benchmark suite and to what extent GCC is able to achieve this. We then compare our approach against GCC's and a start-of-the-art machine learning schemes. This is followed by a brief analysis of the results.

6.6.1 Maximum Performance Available: evaluating GCC's heuristic

In order to determine how well our technique and others perform, we first conduct a limit study. As described in section 6.4 we exhaustively enumerated loop unroll factors up to 15 for each loop, recording the best setting for each. We then ran each benchmark with the best unroll factors set and recorded the speedup. The bars labeled oracle in figure 6.8 show the maximum achievable speedups compared to no unrolling for the benchmarks.

What is immediately obvious is that the impact of loop unrolling varies dramatically across benchmarks with an average speedup of 1.05. For some benchmarks such as `adpcm` from *MediaBench* no unroll factor has an impact on performance. In the case of `security_sha` from *MiBench*, however, there is a potential speedup of 1.28. What

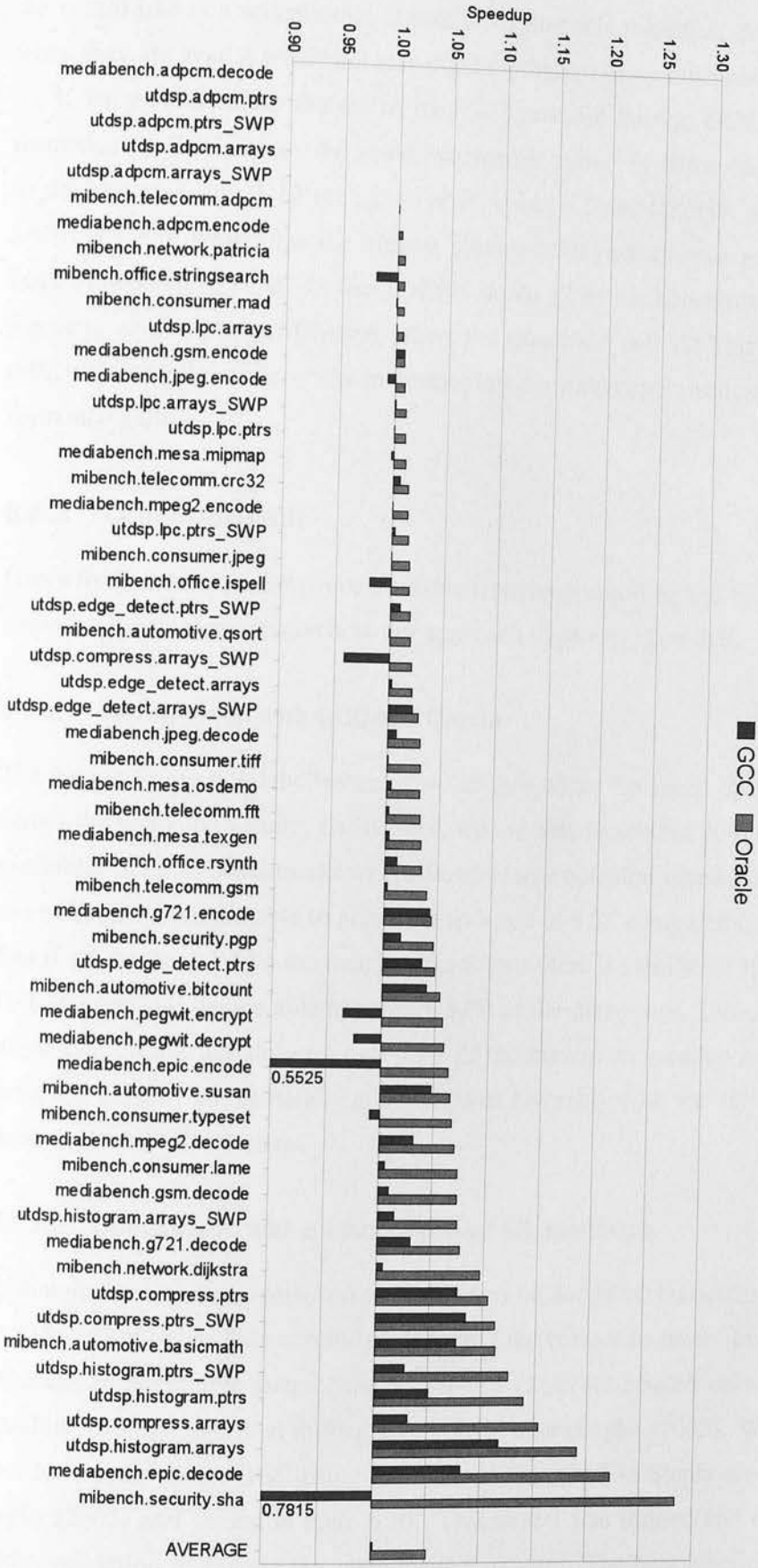


Figure 6.8: Speed up of the unroll factors chosen by GCC's default heuristic and the speed up of the best possible unroll factor - the oracle.

we would like is a scheme that is able to exploit this potential: delivering speedups when they are available and not slowing the program down otherwise.

If we now consider the set of bars in figure 6.8 labeled GCC, we see the performance of GCC across the same benchmark suite. In some case cases it is able to achieve speedup, 1.12 on `histogram.arrays` from *UTDSP*, yet in the case of `security_sha` which has the biggest potential for performance gains, it delivers a large slowdown of 0.78. In fact it slows down 12 of the benchmarks, the worst being `epic_encode` from *MiBench*, where the slowdown is 0.55. This demonstrates the difficulty compiler writers have in developing a portable optimisation that delivers performance gains.

6.6.2 Our Approach

Given the potential performance available from loop unrolling and GCC's poor performance, we here demonstrate how our approach improves upon that.

6.6.2.1 Comparison with GCC and Oracle

The bars in figure 6.9, labelled `Our Technique`, show the speed-ups of our approach across the benchmark suite. On average, we are able to achieve 76% of the maximum available. In those benchmarks where there is large potential speedup available such as `security_sha` we are able to achieve a speedup of 1.21 compared to GCC's 0.78. In fact if we concentrate on the benchmarks where there is significant speedup available ($>$1.10 speedup) we are able to achieve 82% of the maximum. Thus, we have a technique that on average delivers over 75% of the maximum speedup available. This is achieved entirely automatically and compares favorably with the 3% achieved by the hand-crafted GCC heuristic.

6.6.2.2 Comparison with a state-of-the-art ML technique

Although our technique performs well, this may be due to the particular machine learning algorithm rather than carefully generating the correct features. In this section we evaluate an alternative state-of-the art scheme (`StateML`) based on a support-vector machine (SVM) described in Stephenson and Amarasinghe (2005). We implemented this technique within GCC using the features described in Stephenson and Amarasinghe (2005) and shown in table 6.10. This model was trained and evaluated using cross-validation in exactly the same manner as ours. The bars labelled 'StateML' in

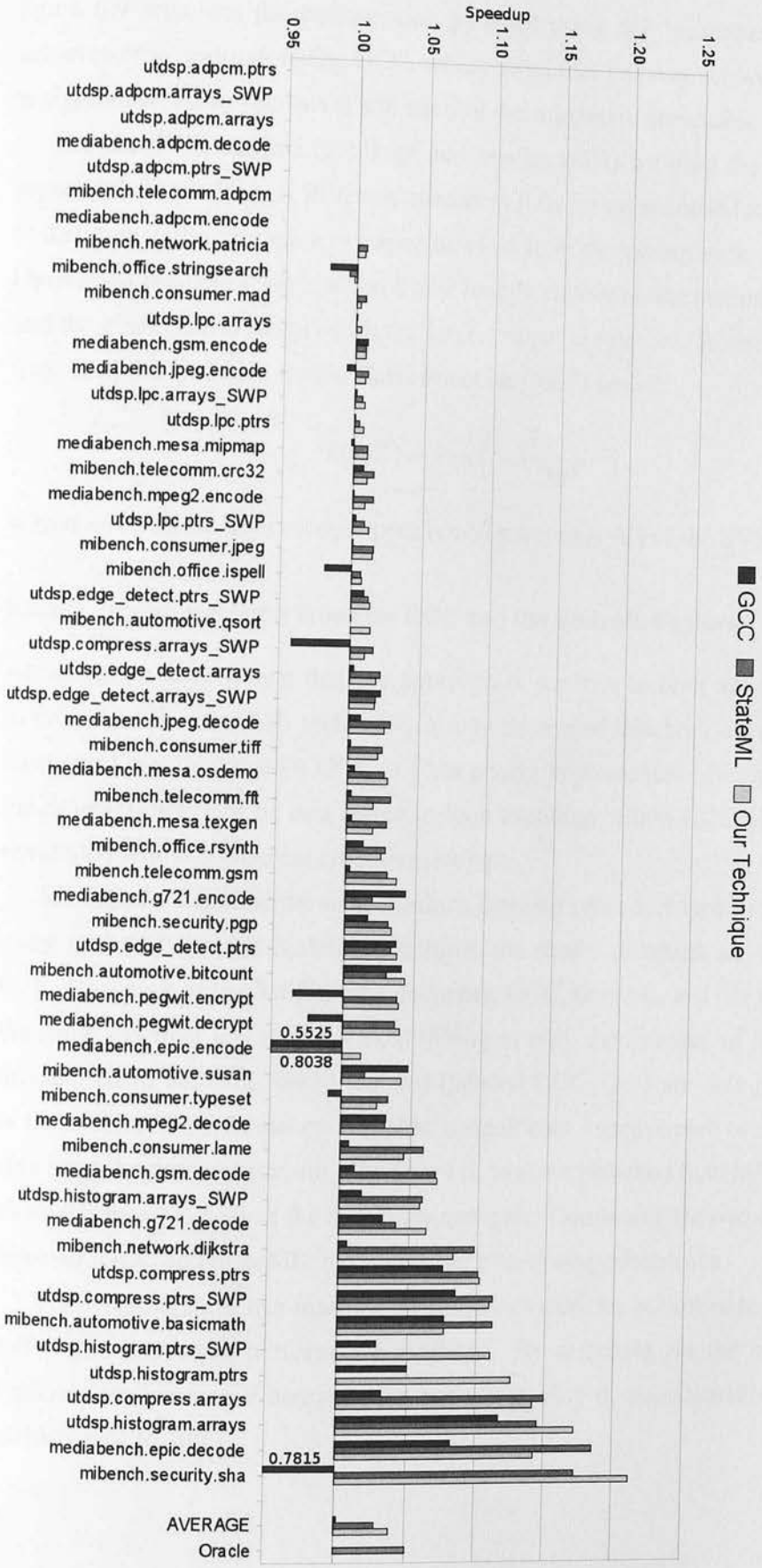


Figure 6.9: Comparison of speed ups between GCC's heuristic (`GCC`) , our technique (`Our Technique`) and the state-of-the-art ML approach (`StateML`). GCC achieves an average of 3% of the performance available, StateML achieves 59% of the performance available while our approach achieves 76% of the maximum performance available.

figure 6.9 represent the performance achieved using this technique. On average it achieves 59%, outperforming GCC, which given that this was achieved automatically is significant. However, this is still short of the maximum achievable.

For the SVM method (Schlkopf and Smola, 2001) we used the "one-vs-all" approach where we learn K different classifiers (one for each unroll factor) each trained to distinguish the examples in a specific class from the examples in all the remaining classes. At prediction time, when a new loop is presented, the classifiers are executed and the class (unroll factor) with the largest ouput is selected. In our experiments we have used the Gaussian Radial Basis Function (RBF) kernel:

$$k(\vec{x},\vec{x}') = \exp\left(-\frac{|\vec{x}-\vec{x}'|^2}{2\sigma^2}\right), \tag{6.1}$$

with $\sigma = 1$ and we have set the upper bound parameter (C) of the SVM to 10.

6.6.2.3 Using Decision Trees for GCC and the StateML Features

Although we have shown that our approach is superior to both the default heuristic in GCC and the StateML technique, it may be argued that both alternative schemes have good features, that (i) GCC just has poorly implemented heuristics and (ii) that the StateML may not be best suited to loop unrolling within GCC given that is was developed within a different compiler setting.

We therefore applied the same machine learning procedure based on decision trees using both GCC' s and StateML's features, the results of which are shown in figure 6.11. Thus each of the 3 different approaches, GCC, StateML and our technique share the same machine learning model, differing in only their choice of features. Using this approach, the GCC based features (labeled GCC Tree) are able to achieve 48% of the maximum performance available, a significant improvement over the 3% available from the default heuristic. The StateML features (labelled StateML Tree) slightly worsen from 59 to 53% of the maximum available. Combining the two sets of features, however, GCC and StateML, has no further impact on performance.

These results show that machine learning does work but is limited to approximately half of the maximum performance available. By searching for the best features in tandem with learning a heuristic, we have been able to automatically improve this performance to 76%.

The loop nest level
The number of operations in loop body
The number of floating point operations in loop body
The number of branches in loop body
The number of memory operations in loop body
The number of operands in loop body
The number of implicit instructions in loop body
The number of unique predicates in loop body
The estimated latency of the critical path of loop
The estimated cycle length of loop body
The language (C or FORTRAN)
The number of parallel "computations" in loop
The maximum dependence height of computations
The maximum height of memory dependencies of computations
The maximum height of control dependencies of computations
The average dependence height of computations
The number of indirect references in loop body
The minimum memory-to-memory loop-carried dependence
The number of memory-to-memory dependencies
The trip count of the loop (-1 if unknown)
The number of uses in the loop
The number of definitions in the loop.

Figure 6.10: The StateML features

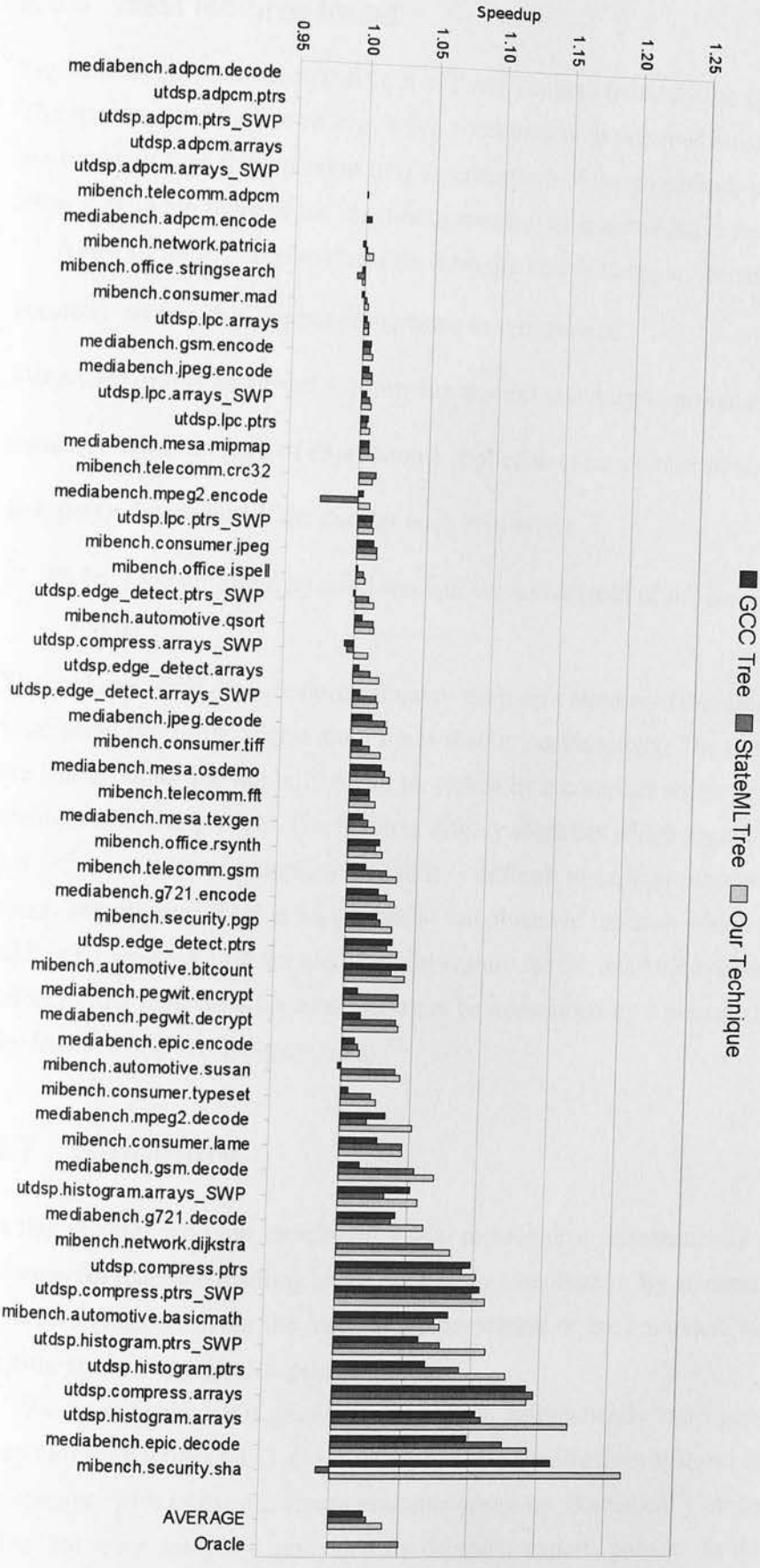


Figure 6.11: Speed-up using a decision tree as the learning technique for GCC's, StateML compared to our technique. Keeping the machine learning algorithm identical shows the relative merits of the feature sets.

6.6.3 Best features found

Figure 6.12 presents the first six out of thirty features found by the system in one fold. The speedup that feature attains, when combined with previous features is given, as is the translation of that speedup into a percentage of the maximum possible. We also show how much more of the maximum speedup each consecutive feature brings.

A few of the expression elements from the best features are explained below.

count(s) returns the number of elements in sequence, s.

filter(s,m) filters sequence, s, removing any not matching expression m

sum(s,e) takes the sum of expression e applied to each member of sequence s

is-type(t) determines if the current node is of type t

/*, //*, [n] the children, descendants and particular child of the current node, respectively.

The first most important feature computes the loop's number of iterations, clearly, there is no point unrolling a loop more times than it has iterations. The remaining features are less obvious and are unlikely to be picked by a compiler writer demonstrating the strength of the approach. The features display elements which appeal to the intuition, but are, nonetheless, complicated and it is difficult to explain exactly why some elements are present. This is an artifact of the objective function which attempts, when adding a feature to find the most helpful feature for the machine learner, but makes no effort to find features whose rationale can be understood by a human. The meaning of the features is given in figure 6.13.

6.7 Summary

In this chapter we have developed a new technique to automatically generate good features for machine learning based optimizing compilation. By automatically deriving a feature grammar from the internal representation of the compiler, we can search a feature space using genetic programming.

We have applied this generic technique to automatically learn good features for loop unrolling within GCC. Our technique automatically finds features able to achieve, on average, 76% of the maximum available speed-up, dramatically outperforming features that were manually generated by compiler experts before. In this chapter our approach focusses on the RTL representation of the loop. The system is generic, however, and is easily extended to cover different data structures within any compiler.

Number	Speedup	% of Max	Improvement	Feature
1	1.01971	38.63%	38.63%	get-attr(@num-iter)
2	1.02665	52.22%	13.59%	count(filter(//*, !(is-type(wide-int) || (is-type(float_extend) && [(is-type(reg))]/count(filter(//*,is-type(int))))) || is-type(union_type))))
3	1.03089	60.52%	8.30%	count(filter(/*, (is-type(basic-block) && (!@loop-depth==2 || (0.0 > ((count(filter(//*, is-type(var_decl))) - (count(filter(//*, (is-type(xor) && @mode==HI))) + sum(filter(/*, (is-type(call_insn) && has-attr(@unchanging))), count(filter(//*, is-type(real_type))))))) / count(filter(/*, is-type(code_label)))))))))
4	1.03353	65.70%	5.18%	max(filter(/*,(is-type(basic-block) && !(@loop-depth==3 && @may-be-hot==true))), count(filter(/*,(is-type(insn) && /[5][(is-type(set) && /[0][(is-type(reg) && !@mode==DF)])]))))
5	1.03448	67.55%	1.86%	count(filter(//*,is-type(array_type)))
6	1.03503	68.62%	1.07%	count(filter(//*,(is-type(le) && !has-attr(@mode))))

Figure 6.12: Best features found by our feature search in one fold

Feature 1 Get the 'number of iterations if constant' attribute of the loop

Feature 2 Count the number of nodes at any depth in the loop which are
neither `wide-int` nodes
nor `float_extend` nodes for which
the first child is not a `reg` node with the count of `int` nodes beneath that `reg` node is zero
nor `union_type` nodes

Feature 3 Count the number of `basic-blocks` for which
the `loop-depth` was not 2 or
(number of `var_decl` nodes in the block minus
the number of `xor` nodes in the block with `mode` attribute `HI` minus
the sum, over all `call_insns` with an `unchanging` attribute of
number of `real_type` nodes
divided by the number of `code_labels` in the block)
is less than zero

Feature 4 For each `basic-block` that is not both of depth 3 and hot, compute
the number of instructions which have child 5 that
is a `set` and its first child is
a `reg` whose `mode` attribute is not `DF`.
Then take the maximum over all `basic-blocks`

Feature 5 Count the number of `array_type` nodes at any depth in the loop

Feature 6 Count the number of `le` nodes at any depth in the loop that do not have a `mode` attribute

Figure 6.13: Meanings of features from figure 6.12.

Chapter 7

Efficiently Generating Training Data

The last two chapters showed how the compiler writer could be freed from having to think about what features to use in their machine learning set ups. This chapter frees the human from one more hurdle; efficiently generating training data.

7.1 Introduction

In iterative compilation, each variation of a program must be run multiple times because of noise in performance measurements; everything from the other processes running on the machine or the state of the file system to the temperature of the computer can have an effect. This becomes more of a problem as the granularity of the measurements becomes finer. When individual functions and loops are measurement targets, the noise to signal ratio can be significant (Monsifrot et al., 2002).

Different approaches are taken to circumvent the noise problem. In some instances, researchers have chosen a fixed sample size plan, running each program version a constant number of times without observing how the results are shaping up as they go (Agakov et al., 2006; Bodin et al., 1998). The hope is that the constant number of runs is sufficiently large to yield good results but not too large to waste effort. Often there is no analysis presented as to whether this number of runs is truly sufficient or if it is too many; confidence intervals and standard error bars rarely feature on performance graphs.

It may be tempting to use simulation to overcome noise, but not only are simulators slow and incompletely accurate, they are also subject to measurement bias Mytkowicz et al. (2009). To overcome that bias, random variations in set up must be effected and simulations run multiple times; the result is noise in the measurements, just as there is

noise for direct execution.

What is needed is a technique which provides statistically rigorous results in the presence of noisy data and simultaneously reduces the cost of searching a large space of optimisation settings. To the best of our knowledge, there is little prior work in this area. The closest to the work in this chapter are Georges et al. (2007); Blackburn et al. (2006) where the authors recommend statistical rigour. They examine each point in the compiler optimisation space in isolation and propose performing executions until an estimate of the sample's inaccuracy is tolerably small or some maximum number of executions is reached. While they attain accurate measurements, they do not reduce the total number of executions needed for the whole optimisation space. Indeed, this chapter will show that their approaches can require more executions than a perfectly selected constant sized sampling plan.

This chapter develops an algorithm which:

- Determines a subset of the optimisation settings or program versions which are 'better' than all others, to some user supplied significance level.
- Minimises the number of runs required, dropping poorly performing versions early and finishing when sufficient data have been gathered for a decision.
- Provides statistically rigorous results.
- Allows more points in the compiler optimisation space to be examined.

Our algorithm 'races' different program versions, allowing those performing poorly to fall by the wayside while their sample size is still small. Those program versions, fighting for the winning position, are allowed more rein, increasing their sample size until either one wins or several draw. We show that our adaptive, sequential sampling plan can dramatically reduce the number of runs needed to find the best program version. In this way, a greater part of the optimisation space can be explored than would otherwise be possible.

The remainder of this chapter is organised as follows. The next section presents examples showing the problem of noisy measurement. Section 7.3 gives an overview of our algorithm for the adaptively managed sampling plan and section 7.4 gives an in depth description of the technique. Then, section 7.5 shows the set up for our experiments, the results of which are given in section 7.6. Finally, section 7.7 concludes the chapter.

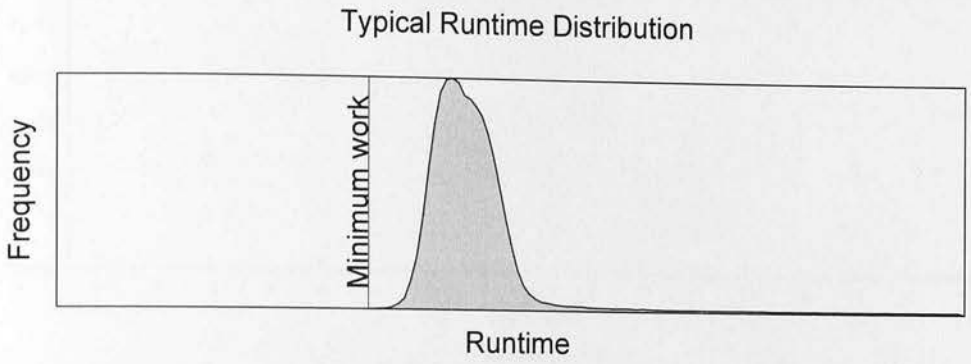


Figure 7.1: Typical distribution for the run time of a program. All programs have a minimum amount of work to achieve, giving a lower bound to the distribution. The shape of the distribution may vary but very often long tails can be observed. If small samples include observations from the tail then means can be thrown off. Without statistically rigorous techniques such situations will not be detected.

7.2 Motivation

Performance measurements on real systems are invariably noisy, an issue which becomes more problematic as the granularity of the measurement becomes finer. Figure 7.1 shows a typical distribution of cycle counts taken from function `run_length_encode_zeros` in the `MediaBench epic-encode` program. There is a minimum amount of work that the program must do, so there is a lower bound to the run time. On the other hand, there is no clear upper bound and the distribution features a long tail. The long tail of the distribution extends well beyond the median point and outliers from that tail, if included, can throw out the mean of small samples.

The difficulty in doing experiments with performance measurements is deciding how many observations are needed for each sample so that a confidence interval around the mean is sufficiently small. Typical, fixed size sampling plans require that this number of observations be fixed before any data is actually gathered and before any estimates of the noise are available.

The next section shows an example in which confidence intervals can prove or disprove the adequacy of different sample sizes.

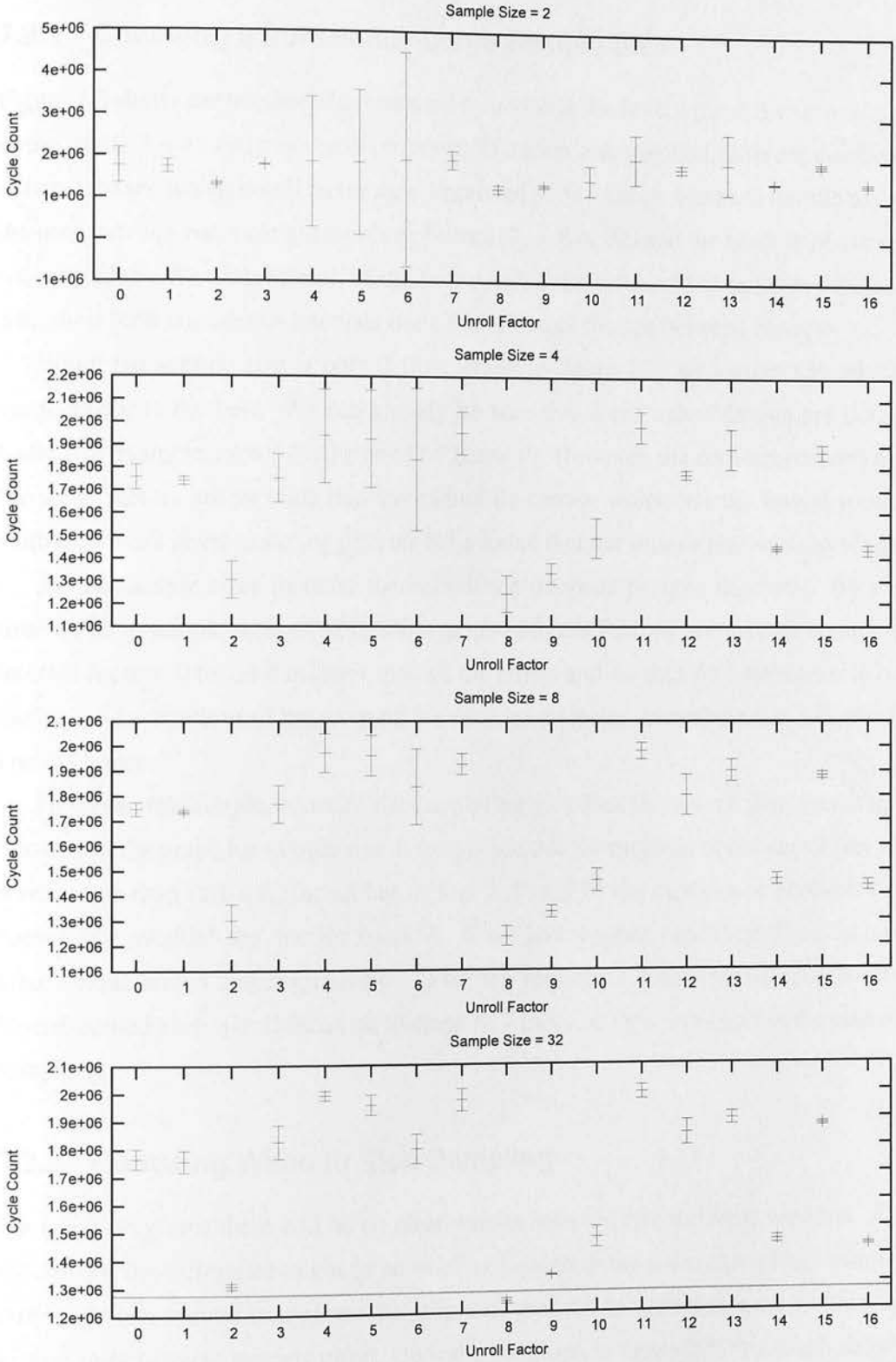


Figure 7.2: Confidence intervals at different sample sizes. Data is from function `run_length_encode_zeros` in `MediaBench epic-encode`. Cycle counts are shown for different unroll factors of a particular loop. Only when the sample size is 32 per unroll factor does an unambiguous winner emerge. The confidence interval is 95%. Note that the cycle count axis changes in each sub-figure.

7.2.1 Choosing a Sufficiently Large Sample Size

Figure 7.2 shows the number of cycles used by a loop in the function `run_length_encode_zeros` in the `MediaBench epic-encode` program. The loop was unrolled different numbers of times, to see which unroll factor most improved performance. For each unroll factor, the program was run a certain number of times (2, 4, 8 or 32) and the number of cycles was recorded. We plot, in each of the four graphs, the mean of the samples together with their 95% confidence intervals (note that the axes change between figures).

When the sample size is only 2 (first graph in figure 7.2) we cannot say which unroll factor is the best. We can already be sure that some unroll factors are doing badly (for example, factor 4 is bettered by factor 9). However, the confidence intervals for some factors are so wide that we cannot be certain which has the lowest mean. With a constant sized sampling plan we have found that our sample size was too small.

As the sample sizes increase the confidence intervals become narrower. By the time we have sample sizes of 32 (bottom graph in figure 7.2), we see that the complete interval for unroll factor 8 is lower than all the others and we thus find that factor to be the best. 32 executions of the program for each unroll factor are sufficient to tell which one to choose.

However, this simple, constant sized sampling plan does more work than necessary. Looking at the graph for sample size 4, we can see that the majority of the unroll factors were worse than factor 8; for all but factors 2, 9 and 16 the confidence intervals lay completely outside[1] the one for factor 8. If we had stopped executing those factors after sample size 4 and continued to 32 for the remaining 4 factors, we would only have executed each unroll factor an average of 7 times, a 78% reduction in the cost of sampling.

7.2.2 Choosing When to Stop Sampling

For some programs there will be no clear winner between two different versions. Alternatively, the difference might be so small compared to the noise that a huge sample size might be required to separate the program versions. In such a case we would like to stop early to avoid wasting effort. Consider the graph in figure 7.3; this graph shows the 95% confidence intervals of a different loop in `MediaBench epic-encode`. Again, the loop is unrolled different amounts, from 0 to 16, but this time the sample size is

[1] We do not advocate performing statistical tests visually in this fashion. Rather, bona fide Student's t-tests, ANOVA, etc. should be used.

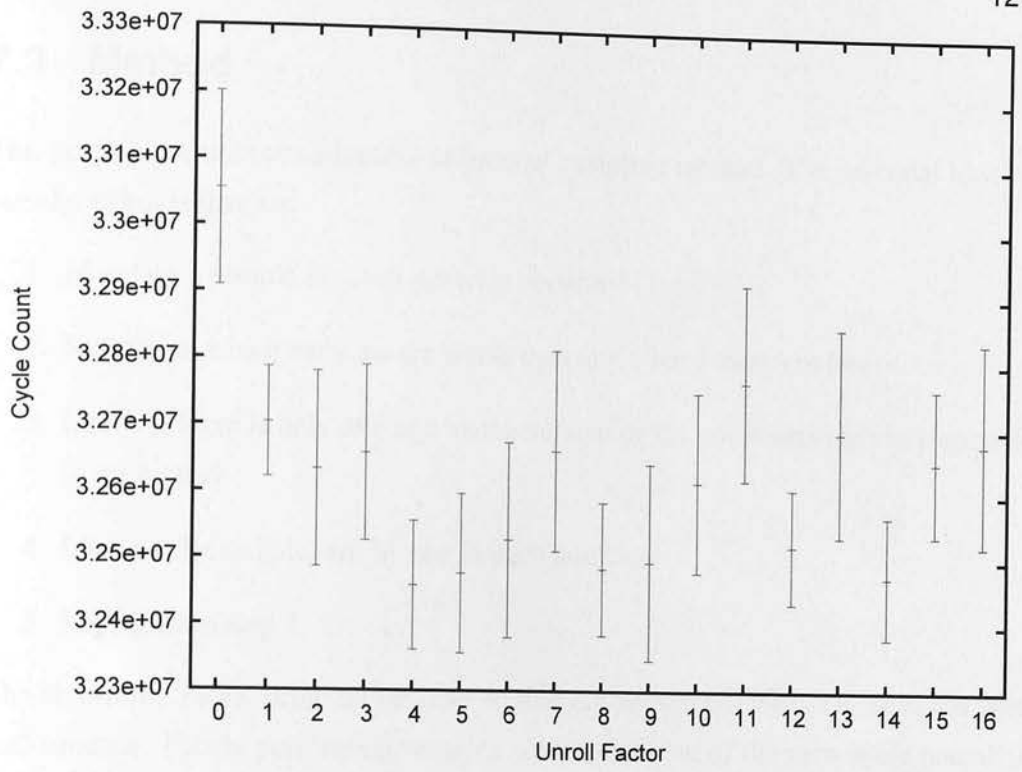


Figure 7.3: 95% Confidence intervals at sample size of 1000 for a loop in function `internal_filter` of benchmark `MediaBench epic-encode`.

1000. Even with this huge increase in sample size, only a few unroll factors (0, 1 and 11) can be excluded because their confidence intervals are completely disjoint to the one for unroll factor 4.

However, if we look at the worst case for unroll factor 4 and compare it to the best case for the other unroll factors we find a ratio of no more than 1.0065. In other words, if we chose factor 4 as the best factor, we could be fairly confident that if we are wrong it would be by not much more than 0.65%. The user, searching for the best program version might consider such a small error acceptable and agree that we need not execute the programs more times.

The situation is likely to be different for each program. In some cases, a small, constant sample size will suffice, in others much larger sample sizes must be taken. The user cannot, in general, know ahead of time how large the sample size should be.

This chapter presents a mechanism by which the sample sizes are adaptively managed to ensure statistically valid results while at the same time drastically reducing the number of executions times needed to select the best program version.

7.3 Method

This section presents our adaptive, sequential sampling method. The essential idea of our algorithm is that we:

1. Maintain a sample for each program version
2. Determine which versions are worse than any other - these are losers
3. Finish if there is only one non losing version or the non losers are close enough to each other
4. Increase the sample size by one in each non loser
5. Repeat from step 2

The algorithm 'races' program versions to find out which will win - i.e. have the best performance. Poorly performing versions are knocked out of the race while potential winners continue, increasing their sample size. The moment we find there is either one clear winner or that the front runners are all good enough, we stop.

Figure 7.4 shows a pictorial example of our algorithm. In the left hand side of the first panel, (a), the samples are initialised for each program version. The right hand side shows the confidence intervals for each sample. Each version is, at this point a potential candidate to be in the winning set. We ensure that enough observations are in each sample for our statistical tests to work since they require a minimum sample size; little can be said statistically about a single observation.

In panel (b) the samples have been tested to see if any can already be identified as clear losers. A number of statistical tests are run (described in section 7.4.3) and any version shown to be worse than one of the other versions is taken out of the race. In the right hand pane, the bottom limits for the confidence intervals of two samples (marked with thin, dotted, red lines) can be seen to be above the top limits of other samples. This indicates that we are confident that those samples are worse than the others and that even if we took more observations we are sure that will not change. The corresponding program versions are removed from the candidate set (becoming no longer shaded in the left hand side of the panel).

Since the remaining set of potential candidates contains more than one version we perform another set of tests to see if the versions are all approximately equal (detailed in section 7.4.4). At this point in the example there is not enough data to call the versions equal.

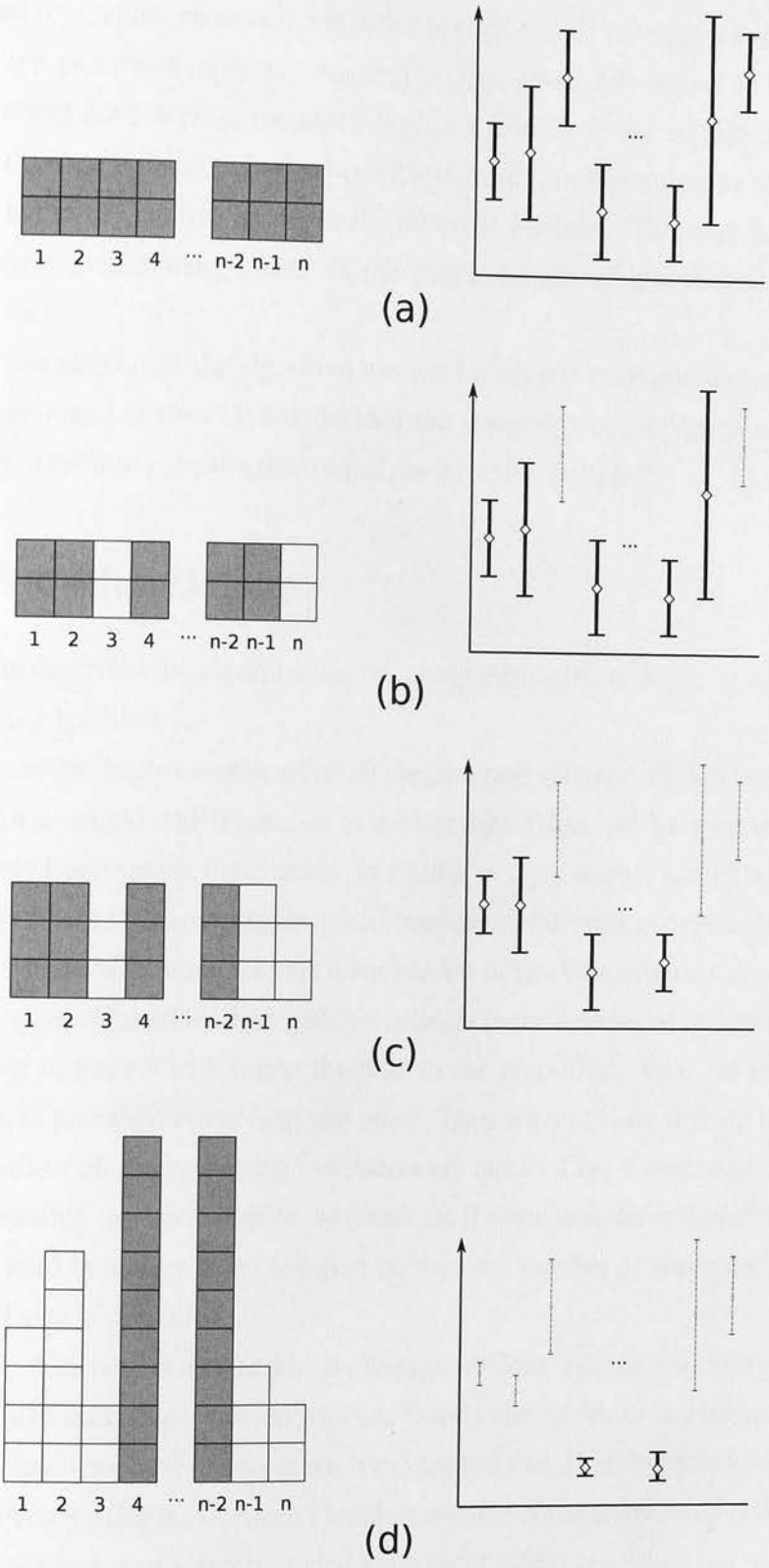


Figure 7.4: Several steps from our algorithm. In (a) initial samples are taken. In (b), versions that are clear losers are dropped. In (c), remaining versions are not deemed equivalent so the samples for them are grown by one. In (d), several steps have been run and the two remaining versions are found to be sufficiently good; the algorithm terminates.

In panel (c) another observation is added to each sample (as shown in the left hand side) and the process is repeated. As sample sizes grow, the amount of information available about each version increases, typically leading to the confidence intervals shrinking (as shown in the right hand side); statistical tests become more able to make decisions about the relative merits of the different versions. This may lead to more program versions becoming losers. In this panel, the second last version is knocked out of the race.

In the last panel, (d), the algorithm has run for several steps and discarded all the program versions but two. It has decided the remaining two program versions are sufficiently similar to consider them equal, so it returns them both.

7.4 Algorithm Details

This section describes the algorithm and its component parts in depth. Pseudo-code is presented in Algorithm 1.

Our algorithm begins with a set of all the program versions, C. For every version we maintain a sample which consists of the run time values we have taken so far for the corresponding version; these are S_c. In reality, we only need to record the sufficient descriptive statistics to determine the mean, confidence intervals and perform statistical tests, we never need to remember the complete list of run time observations and so the amount of space required by the algorithm is linear in the number of program versions.

The loop in lines 3 to 8 forms the bulk of the algorithm. First we remove any version that is provably worse than any other. Then we terminate if there is only one candidate left or all the remaining candidates are equal. Line 6 increases the sample size for remaining versions. Finally we terminate if some user defined limit is reached, allowing a hard boundary to be imposed on the total number of times each program version will ever be executed.

The loop does not remember which program versions were losers from iteration to iteration. This means that a version which is found to be a loser in one iteration has the opportunity to re-enter the race later on. It can happen that a version deemed promising early on turns out to be less so once more information about it has been gathered. We found that reconsidering losers provided lower error rates.

A detailed description of the subroutines used by the algorithm follows.

Algorithm 1 Pseudo code for our algorithm

1. $C \leftarrow \{c; c \text{ is a compilation strategy}\}$ ▷ C is the set of all compilation strategies
2. $\forall c \in C, S_c \leftarrow \{sampleRuntime_c(), sampleRuntime_c()\}$ ▷ minimum sample for each
3. **for ever do**
4. $\quad C' \leftarrow C - losers(C, S)$ ▷ remove any losing versions
5. $\quad$ **if** $|C'| = 1$ **or** $candidatesEqual(C', S)$ **then return** C' ▷ done if one left or all equal
6. $\quad \forall c \in C', S_c \leftarrow S_c \cup sampleRuntime_c()$ ▷ increase the sample size
7. $\quad$ **if** $sampleThresholdReached(C', S)$ **then return** C' ▷ stop if too many samples
8. **end for**
9. $sampleRuntime_c()$ ▷ measure performance and take logarithm
10. $\quad x \leftarrow$ execute strategy c and record runtime
11. $\quad$ **return** $\ln x$
12. $losers(C, S)$ ▷ losers are beaten by some other program version
13. $\quad$ **return** $\{c \in C; \exists d \in C, d \neq c, S_d <_{\alpha_{LT}} S_c\}$
14. $candidatesEqual(C', S)$ ▷ check if all are sufficiently close
15. $\quad B \leftarrow \{b \in C'; \mu_b \leq \mu_c, \forall d \in C'\}$
16. $\quad$ **return** $\bigwedge_{b \in B, c \in C'; b \neq c} S_b =_{\alpha_{EQ}, \varepsilon} S_c$
17. $sampleThresholdReached(C, S)$ ▷ check that no sample is too large
18. $\quad$ **return** $\bigvee_{c \in C} |S_c| \geq MAX\_SAMPLE\_SIZE$

7.4.1 Initialisation

At the beginning of the algorithm the sample sets are initialised to have two observations, the minimum necessary to make statistical inferences with a t-test. If other statistical tests are used, the initial number of observations may have to be different.

7.4.2 Sampling the Run time

The cycle count for the current program version is measured by the function *sampleRuntime*. It assumes that there is some mechanism to profile the program to calculate the measurement.

Of special note here is that we take the natural logarithm of the run time. The reason for this is that run time distributions are both skewed and often suffer from outliers; applying a log transform is a common way to make the distribution look more 'normal' and to reduce the effects of outliers (Bland and Altman (1996)). Having distributions which are closer to a normal distribution frequently improves the accuracy

of statistical tests. More general transformations, such as a Box-Cox transform (Box and Cox (1964)) of which the logarithm is a special case, could be used instead, but we found adequate results from the simple logarithm.

7.4.3 Weeding Out Losers

The algorithm needs to determine which versions are unlikely to be in the final winning set which is handled by the function, *losers*. The set of losers consists of any version, c, for which there is another version, d, that looks to be better performing. The 'better performing' test is a relation, $<_{\alpha_{LT}}$, over samples, (S_d, S_c). Intuitively we want $S_c <_{\alpha_{LT}} S_d$ whenever it appears that the true mean of S_c is worse than the true mean of S_d to some confidence level. This relation is not a total ordering since if we do not have enough observations to be confident, then neither $S_c <_{\alpha_{LT}} S_d$ nor $S_d <_{\alpha_{LT}} S_c$ will hold.

To determine membership of the relation, $<_{\alpha_{LT}}$, we first check that the mean, μ_d, of S_d is less than the mean, μ_c, of S_c. If that is the case then we perform a student's t- test to discover if the difference in the means is significant to some user supplied significance level, α_{LT}.

Since we cannot be certain that the variances are equal and since also the number of observations in each sample may be different, we use Welch's t-test (B.L.Welch (1947)) as described in 2.6.3.

As the testing proceeds, any version removed is not used to compare against subsequent versions in the tests for this iteration. In this way the algorithm guarantees that at least one version survives the *losers* function.

7.4.4 Finding the Winners

Stopping when enough data have been gathered is important since we may find that some program versions are either identical to each other or so nearly so that the compiler writer is content with several winners. Increasing the sample sizes after this point will waste effort and, left unchecked, may continue indefinitely.

In the happy case that one version has beaten all others, we can return that single winner. If more survive then we check to see if they are all sufficiently close together for the compiler writer, decided by function *candidatesEqual*. This function performs a statistical equivalence test, not using Westlake intervals since that would require the indifference region to be specified as a difference of an absolute number of cycles. It is more natural, instead, to describe the indifference region as an upper bound on the ratio

of means. Our justification for this is that speed ups and slow downs are important in compiler fields, but rarely are absolute cycle counts. We might consider a speed up of 1.001 to be uninteresting, regardless of whether it represents one million or ten billion cycles; conversely a speed up of 1.5 is likely exciting with similar disregard for the number of cycles in the difference.

We expect compiler writers to be interested in the version returned with the lowest mean (the rest of the set we expect to be only of cursory interest, except perhaps to machine learning tools). We use this information to tune our equivalence test to the problem. Since the compiler writer will choose the one from the non losing set with the lowest mean, we wish to ensure, to some confidence, that none of the other non losers could have provided much of a speed up compared to that choice.

We can determine the most available speed up between two program versions by taking a confidence interval for each sample:

$$upper = \mu + t_{(\alpha_{EQ},n-1)}\sqrt{s^2/n}$$

$$lower = \mu - t_{(\alpha_{EQ},n-1)}\sqrt{s^2/n}$$

where t is the student's t upper percentage point value for a given user supplied probability, α_{EQ}, and μ, s, and n are as before.

Now, if the program version with the lowest mean is b, and we have another version, c, we can calculate the worst case speed up of c over b by comparing the upper end of b's confidence interval against the lower end of c's interval. However, we must remember that we initially transformed our observations with a logarithm transform. If we simply compare the interval ends directly we will not be describing speedups; we must apply the inverse transform, first (note that it is this transformation which prevents us using Fieller's theorem (E.C.Fieller (1954); M.A.Creasy (1956)) for the confidence interval of the ratio of two means). The speedup is:

$$worstspeedup_{b,c} = \frac{e^{upper_b}}{e^{lower_c}} = e^{(upper_b - lower_c)}$$

This shows us the speed up in the worst case where the true value of the mean for version b is at the upper end of its confidence interval and the true mean for c is at the lower end of its interval.

The compiler writer, having already given a significance level, α_{EQ}, must also now specify an indifference region, ε. This gives the maximum worst case speed up for the equivalence test. Putting this together, we now have a relation, $=_{EQ,\varepsilon}$, used in line 16,

over pairs of samples such that:

$$(S_b, S_c) \in =_{EQ,\varepsilon} \leftrightarrow 1 + \varepsilon < worstspeedup_{b,c}$$

Other stopping conditions are possible, but we believe that this estimate of the nearness of the run times closely matches the requirements of iterative compilation. If the current experiment at hand needs a different stopping condition it should be easy to adjust our algorithm to it.

7.4.5 Limiting Total Sample Size

Our algorithm also gives the compiler writer the opportunity to place hard limits on the total sample size through the constant, *MAX_SAMPLE_SIZE* in function *sampleThresholdReach*. This fixed limit makes ours a restricted sampling plan (Wetherill and Glazebrook (1986)); other methods exist to ensure closed sample boundaries (P. and M.J.R. (1957)) and may be worth considering, although we have found that the restricted plan is quite adequate.

Having a hard sample size limit allows the compiler writer to choose how keen they are for correct results. A large limit permits the algorithm to expend more effort disambiguating difficult cases. Feedback is given when the limit is reached so that in those cases the compiler writer can either accept the best estimate so far from the algorithm or reject, deciding that further effort is not warranted. With a combination of this limit and the two significance levels, the compiler writer can tune the breadth and accuracy of the space they wish to explore.

7.5 Experimental Setup

In this section we briefly describe the experimental set up. We performed two different experiments; the first was to find the best unroll factor for loops, while the second was to find the best compiler flags for whole benchmarks.

7.5.1 Experiments

7.5.1.1 Loop Unrolling Experiment

Loop unrolling has been targeted in a number of previous machine learning and iterative compilation works (Monsifrot et al. (2002); Stephenson and Amarasinghe (2005)).

Being a fine grained optimisation, there is a wide range of variation in noise to signal ratios in different loops.

7.5.1.2 Compiler Flags Experiment

Finding the best compiler flags has also been widely explored in both iterative compilation and machine learning (Fursin et al. (2008a)). The very long data gathering phases, often equating to months of compute time (Fursin et al. (2008a)), make efficiency of paramount importance.

7.5.2 Compiler Setup

For both experiments we used GCC 4.3.1. In the first we extended the compiler to allow unroll factors to be explicitly specified for each loop in a program. In the second we altered GCC to accept command line arguments externally, regardless of the benchmarks' makefile. This allowed us to force different compilation flags to be used.

7.5.3 Benchmarks

7.5.3.1 Loop Unrolling Experiment

For the loop unrolling experiment we took 22 embedded benchmarks from the MediaBench and UTDSP benchmark suites. Those benchmarks which did not compile immediately, without any modification except updating path variables, were excluded.

7.5.3.2 Compiler Flags Experiment

For the compiler flags experiment we added the MiBench suite, extending the number of benchmarks to 57. Again, we excluded those which did not immediately compile.

7.5.4 Platform

These experiments were run on an unloaded, headless machine; an Intel dual core Pentium 6 running at 2.8 GHz with 2Gb of RAM. We used a fast machine so that we could gather sufficient data for the naïve, constant sized sampling plans.

7.5.5 Data Generation

7.5.5.1 Loop Unrolling Experiment

For loop unrolling, we selected each of the 230 loops in the benchmarks and unrolled them different number of times. Each loop was unrolled between 0 and 16 times inclusive, giving a total of 17 different program versions per loop. Only one loop was modified at any one time, meaning that a program with ten loops would be compiled 170 times. This allowed each loop to be considered in isolation.

The current loop being changed was instrumented to record the cycle count before and after the loop. Each different version of a program was run 1000 times.

7.5.5.2 Compiler Flags Experiment

For the compilation flags experiment, we took 86 of GCC's flags and generated random collections of them. Each flag had a 5% chance of being set in each collection. Each benchmark was compiled using the flags from each new collection. If that produced a different binary than had already been seen for the benchmark, then the binary was run at least 100 times with the cycle count recorded.

On some benchmarks, particularly small ones, the compiler would generate identical binaries for many different flag collections. Thus, the number of different points in the compiler optimisation space varied across the benchmarks, from 288 for `mediabench.pegwit` to 34 for `mibench.telecomm.adpcm`.

7.5.6 Failure Rate

We need a method to compare the different sampling plans. We want to ensure that the result of a sampling run chooses a program version which, if it is not the best, is slower than the best by no more than a given amount. If the version chosen by a sampling run is further from the best, then we call the run a failure, else we call it a success. This allows, by repeated running of the sampling plan, to generate mean failure rates and hence compare the quality of different plans; a better plan will have a lower mean failure rate.

Specifically, we desire that the true mean of the performance of whatever program version is finally selected by a plan should be sufficiently near to the true mean of the best possible version. The true mean is estimated by taking the mean of the full data set. We distinguish the estimated true mean as $\overset{*}{\mu}_i$ for the i^{th} program version, as

opposed to the sample mean, μ_i.

If some sampling plan selects a version, c, and the best possible version according to the complete data set is b, then we compare the ratio of the estimated true means, $\hat{\mu}^*_b/\hat{\mu}^*_c$, which gives the slowdown caused by choosing version c over version b. If the ratio is greater than $1-\theta$, for some positive θ, then we call the trial a success, otherwise it is a failure. In our experiments we fix θ to 0.5%, which means that, to be successful, a sampling run must choose a program version whose true mean is no more than 0.5% slower than the true mean of the best possible version.

7.5.7 Techniques Evaluated

7.5.7.1 Profiled Races

To evaluate our approach, we explored different values of the parameters which define our algorithm. Both the significance level for the less than test in the *losers* function, α_{LT}, and $alpha_{EQ}$, the significance level for the equality test in *candidatesEqual*, were allowed to range over the set {0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5}. The set of significance levels gives a broad spectrum of significances from extremely high significance at 0.0001 to very low significance at the other end. For example, when $\alpha_{EQ} = 0.5$, the confidence intervals for two samples, even small ones, will likely be very narrow, so the algorithm will be very generous considering if two versions are equivalent. Conversely, if $\alpha_{LT} = 0.0001$, then the confidence interval for two samples will be wide unless either the samples are large or there is a very small standard deviation; the algorithm will be very sure before deciding that one program version outperforms another.

In all cases, the threshold, θ, which determines how far from the best is acceptable in function *candidateEqual*, is set to 0.5%, matching the boundary for failure. The *MAX_SAMPLE_SIZE* constant is set to the maximum amount of data available in each experiment; in day-to-day use this constant may not be so large. Each parameter setting was run 100 times to determine the mean failure rate and average sample size for those values.

7.5.7.2 Constant Sized Sampling Plan

The first technique we compared against is a straight forward constant sized sampling plan. Here a fixed number of observations is taken of each program version's runtime

or cycle count. Again, we ran plans for each sample size 100 times to generate mean failure rates.

7.5.7.3 JavaSTATS

The second method for comparison is the statistically rigorous approach, JavaSTATS (Georges et al. (2007); Blackburn et al. (2006)). JavaSTATS runs each program version until an estimate of the sample's inaccuracy is sufficiently small. The inaccuracy metric is a confidence interval divided by the sample mean. This metric provides a unit-less indicator of the accuracy of the current sample; a value near to zero is an accurate sample. As the sample size grows to infinity the metric generally approaches zero.

Several parameters are required: α, the significance level for the confidence intervals; $theta$, the threshold at which the metric indicates an accurate sample and minimum and maximum sample sizes. We allowed the significance level and threshold to both range over the set $\{0.0001, 0.005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$. The minimum and maximum sample sizes were set at 2 and the maximum size of the data respectively, just as for our own algorithm.

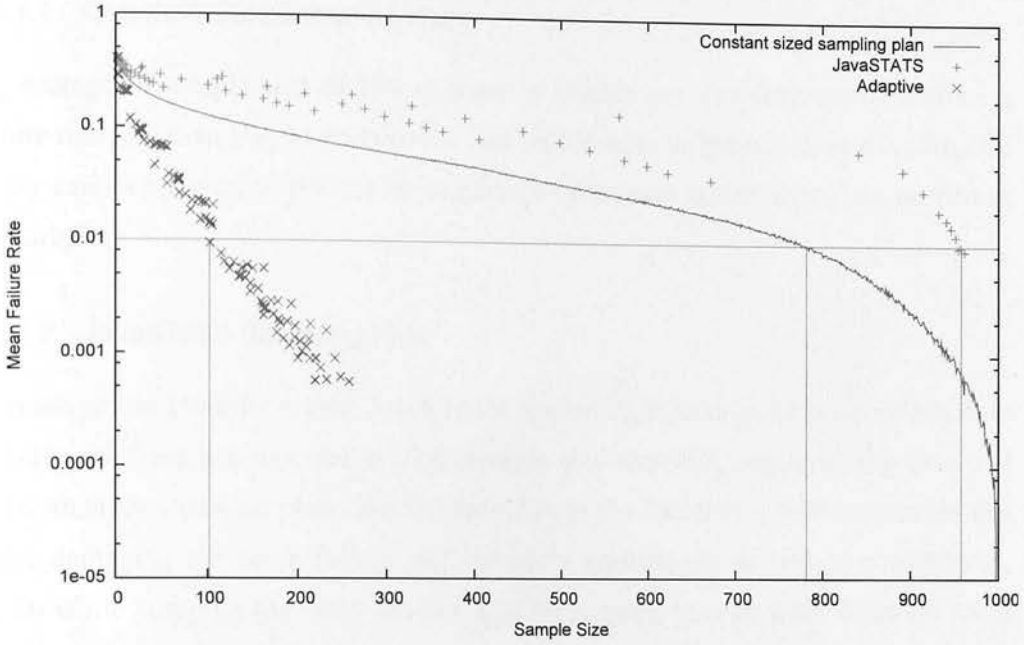
7.6 Results

For each of our two experiments we compared average sample sizes and mean failure rates of the different techniques with their different parameter values.

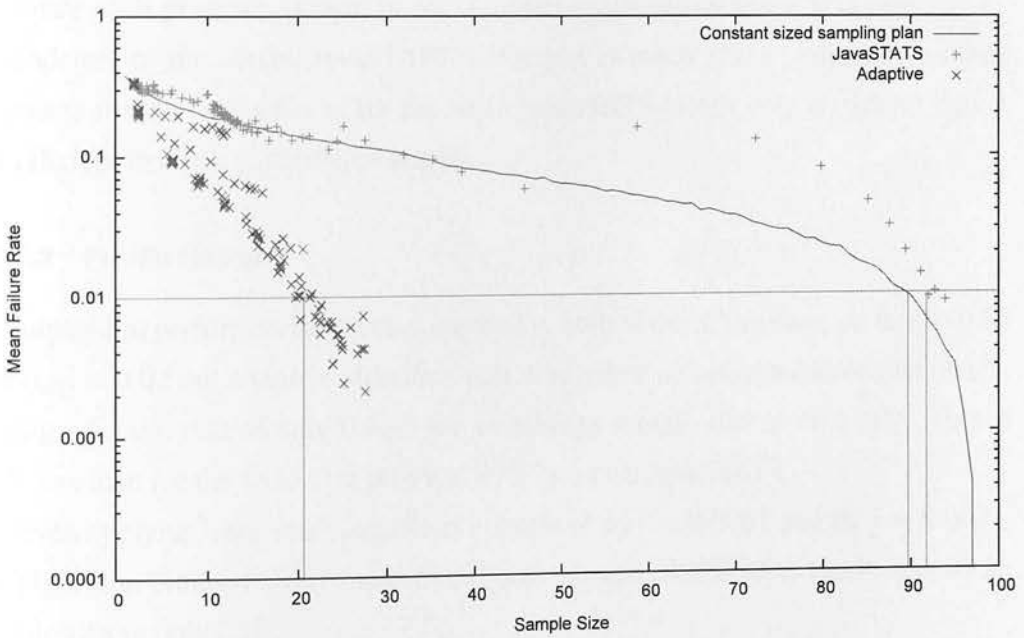
7.6.1 Loop Unrolling Experiment

Figure 7.5(a) shows the performance of the different techniques for the loop unrolling experiment. A horizontal line indicates a 1% mean failure rate, an arbitrarily chosen point of comparison. Mean failure rates are shown against average sample sizes. A failure is when the given number of samples fails to find a program version that is within 0.5% of the cycle count of the best version. Intersection with a failure rate of 1% is shown.

The many points of our adaptive sampling plan show results with different values of α_{LT}, the significance level for the *losers* function, and α_{EQ}, the significance level for the $candidatesEqual$ function, for these, θ, the equivalence threshold, is always 0.5%. Different points from the JavaSTATS algorithm are also shown with varying α and θ.



(a) Loop Unrolling. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ needed only 102 samples compared to an optimal fixed sample plan of 780 samples, a reduction of 87%. JavaSTATS required 956 samples to dip below the 1% failure rate, our reduced that by 89%.



(b) Compilation Flags. At the 1% failure rate, an adaptive plan with $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ needed only 21 samples compared to an optimal fixed sample plan of 90 samples, a reduction of 76%. JavaSTATS required 92 samples to dip below the 1% failure rate, our reduced that by 77%.

Figure 7.5: Comparison between our method, constant sized sampling and JavaSTATS (Georges et al. (2007)).

7.6.1.1 Constant Size Sampling Plan

On average, a sample size of 780 or more is needed per unroll factor to achieve a failure rate less than 1%. In practice the size will have to be greater since the compiler writer cannot have prior, perfect knowledge of how many observations are needed in the samples.

7.6.1.2 JavaSTATS Sampling Plan

To manage the 1% failure rate, JavaSTATS needed tight settings of $\alpha = 0.0005$, $\theta = 0.0001$. At these settings, the average sample size was 957, nearly all the data and worse than the constant plan. We attribute this to the fact that it will sometimes fail early, damaging the mean failure rate; punitive settings are needed to compensate, which while stopping the early failures also force large sample sizes when no early failure has occurred. On the other hand, with such aggressive settings, the compiler writer gets good results without perfect knowledge of the right sample size as is required for the constant sampling plans.

Since each program version in the compiler optimisation space is considered independently of the others, JavaSTATS will spend as much effort producing accurate estimates for the poor ones as for the best. JavaSTATS brings only statistical rigour, not efficient iterative compilation search.

7.6.1.3 Profile Races

Our algorithm performed very well compared to both of the other plans. At $\alpha_{LT} = 0.02$ and $\alpha_{EQ} = 0.02$ our adaptive algorithm first dips below an average failure rate of 1%, getting a failure rate of only 0.94% for an average sample size of only 102. This is 87% less than for the fixed size plan and 89% less than JavaSTATS.

Even applying very strict significance levels of $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, our algorithm attained a tiny mean failure rate of just 0.056% for a small increase in sample size to only 257.

7.6.2 Compiler Flags Experiment

Figure 7.5(b) shows the performance of the different techniques for the compiler flags experiment. The graph looks very similar to that of the loop unrolling experiment.

7.6.2.1 Constant Size Sampling Plan

To achieve a failure rate less than 1%, an average sample size of 90 was needed per program version.

7.6.2.2 JavaSTATS Sampling Plan

Once more, JavaSTATS did slightly worse than the constant plans. At settings of $\alpha = 0.001$, $theta = 0.0001$ the failure rate dipped below 1% with an average sample size of 92. Again, it is still better to use JavaSTATS than the constant plan since it does not require knowing the perfect sample size ahead of time.

7.6.2.3 Profile Races

At $\alpha_{LT} = 0.002$ and $\alpha_{EQ} = 0.01$ our algorithm gets a failure rate of less than 1%, needing only 21 observations on average in each sample. This is 76% less than the constant plan and 77% less than JavaSTATS.

Again, cautious use of very strict significance levels, $\alpha_{LT} = 0.0001$ and $\alpha_{EQ} = 0.0001$, is not costly. With these values, our algorithm needs a sample size of just 27 and gets a mean failure rate of 0.021%.

7.6.3 Parameter Sensitivity

Figure 7.6 shows a few contours for different fixed values of the α_{LT} and α_{EQ} significance levels, demonstrating the role those parameters play in controlling the sample size and failure rates. The contours in the figure are for the loop unrolling experiment, but are very similar to those in the compilation flags experiment.

The points with the highest mean failure rate are those with high values of α_{LT} and α_{EQ}. When these are 0.5, for example, the algorithm never increases the samples more than the minimum since at that level confidence intervals are very narrow. The points with the lowest failure rate are those where α_{LT} and α_{EQ} are also the smallest, as is expected since they demand more confidence before either discarding versions or determining equality.

7.6.4 Individual Cases

In the previous section we showed that our adaptive algorithm significantly outperforms both a simple, constant sized sampling plan and the more complex JavaStats.

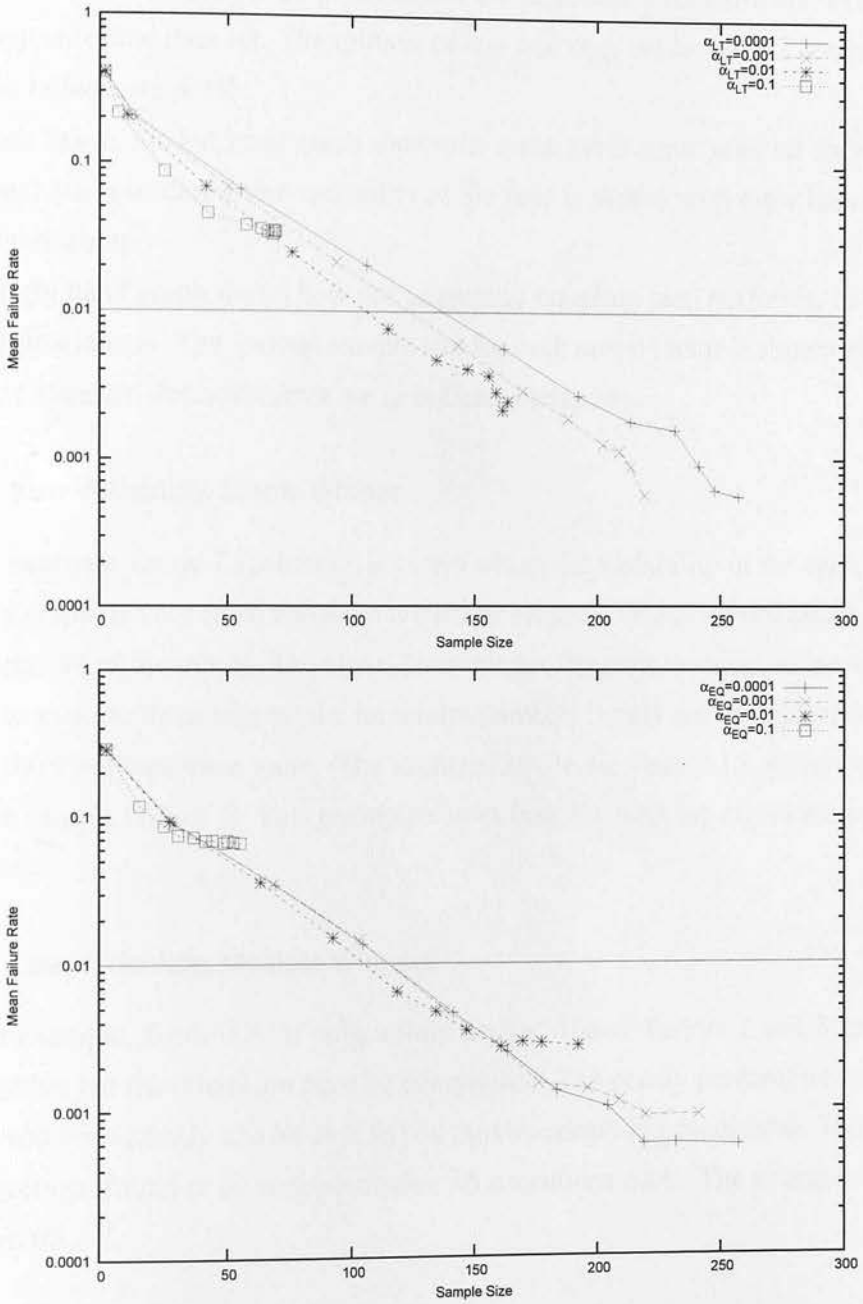


Figure 7.6: Parameter sensitivity. Contours of the two significance levels, α_{LT} and α_{EQ}, are shown for the loop unrolling experiment in figure 7.5. α_{LT} is the the significance level for the less than test in the *losers* function; α_{EQ} is the significance level for the equality test in $candidatesEqual$

In this section we show how our sequential sampling plan performs on a number of individual cases, giving a flavour of what to expect.

Figures, 7.7 to 7.10, show the behaviour of our technique over particular examples of the loop unrolling data set. The settings of α_{LT} and α_{EQ} are both 0.02, the point at which the failure rate is 1%.

In each figure, the left hand graph shows the mean cycle count after all data (1000 executions) are considered; the variability of the data is shown with error bars at one standard deviation.

The right hand graph shows how our sequential sampling plan performs, averaged over 100 simulations. The average sample size for each unroll factor is shown together with a one standard deviation error bar to indicate variability.

7.6.4.1 Low Variability, Single Winner

The first example, figure 7.7, shows a scenario where the variability in the cycle count (left hand graph) is very small and there is a single program version which significantly outperforms all of the others. The algorithm excludes the poor versions, often without needing to execute them beyond the minimum number; it only needs to execute from the best three perhaps once more. The average sample size was 2.15, compared to a minimum sample size of 2. This example shows how the well the algorithm handles easy cases.

7.6.4.2 Low Variability, Multiple Winners

The next example, figure 7.8, is only a little harder. Unroll factors 1 and 3 are very close together but the others are poor by comparison. The poorly performing versions are removed very quickly and focus is left on the two remaining candidates. These two are, on average, found to be equivalent after 35 executions each. The average sample size was 6.05.

7.6.4.3 High Variability, Multiple Winners

In figure 7.9, the different versions are more difficult to distinguish. The algorithm needs larger samples to come to a conclusion, but still is able to reduce efforts on poorer versions. For this problem, average sample size was 161.8 with no failures in 100 simulations. On average, the algorithm returned 8 of the 17 unroll factors in the winning set.

7.6.4.4 Sample Exhaustion

Finally, figure 7.10, shows a much harder case. Here some of the versions could not be culled and, at the same time, could not be proved to be equal. In 97% of the sampling runs the algorithm terminated because the maximum sample size limit was reached. The average number of samples was 454.9 and again there were no failures.

7.7 Summary

In this chapter we have shown that using fixed sized sampling plans can have unintended consequences for performance measurement and iterative compilation. Too small a sample can generate incorrect results. Noisy data can make a small sample appear to have a promising mean where a larger sample would give a very different answer. It is not enough to only look at the means of performance measurements, the statistical significance of the results must be taken into account.

Too large a sample wastes excessive work since program versions are executed an unnecessary number of times. Often, some program versions could be discarded early but a fixed sampling plan is oblivious to these opportunities. With fixed sampling plans, the user must choose, ahead of time, how many runs each version needs to get good data. This is very difficult to do without already having data to examine.

We have provided an algorithm which automatically adapts to the requirements of the problem at hand. In cases where there is little noise and a clear winner is visible early the algorithm will take very few samples. When particular versions require larger sample sizes to disambiguate them the algorithm does just that. Finally, the algorithm terminates when versions are equivalent so that no further work is done trying to tell identical versions apart.

We applied our technique to finding the best loop unrolling factor for a number of loops from different benchmarks and also to finding the best compiler flags for whole programs. Some loops and programs generated noisy data while others had relatively clean data. Our method was able to adapt to these differences, choosing different sample sizes in each case. We reduced the cost of iterative compilation by between 76% and 87% compared to a fixed sized sampling plan. Compared to JavaSTATS (Georges et al. (2007)), we reduced the cost by between 77% and 89%.

Our technique allows the data gathering phase of machine learning in compilers

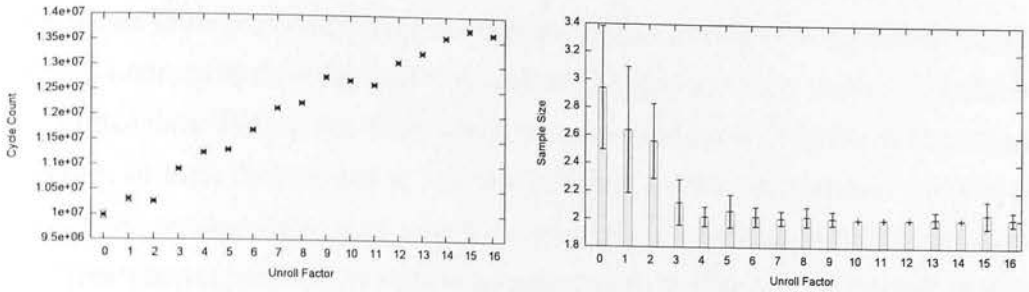


Figure 7.7: A loop from `MediaBench gsm-encode`, function `Gsm_preprocess`. If the winner is clear very early on, then very small sample sizes will result.

Error bars are, in both graphs, one standard deviation.

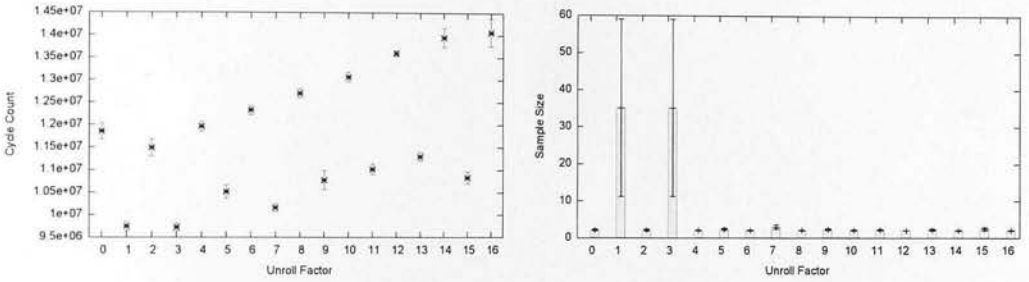


Figure 7.8: A loop from `MediaBench adpcm-decode`, function `adpcm_decoder`. Poorly candidates are quickly discarded and effort focused on the remaining set.

Error bars are, in both graphs, one standard deviation.

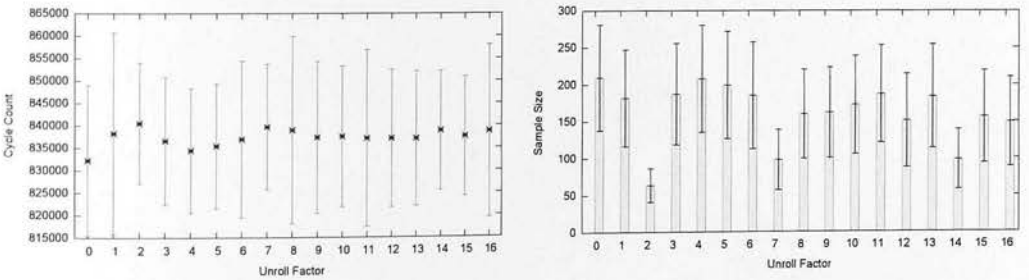


Figure 7.9: A loop from `MediaBench jpeg-encode`, function `emit_eobrun`. When the relative noise is large more samples must be taken.

Error bars are, in both graphs, one standard deviation.

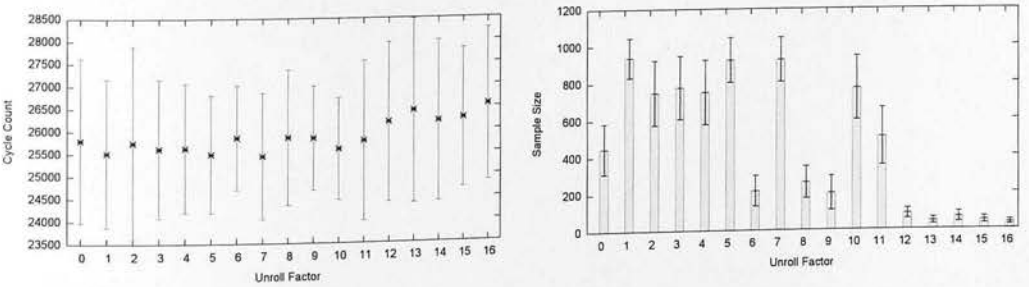


Figure 7.10: A loop from `UTDSP fft_1024`, function `main`. Sometimes the noise will case the sample limit restriction to be reached.

Error bars are, in both graphs, one standard deviation.

experiments to be performed without compiler writers having to worry about the statistical soundness of their data and without having to pay any more than strictly necessary for that data. Before this work, compiler writers had either to ignore the statistical properties of their data or had to run each program version an excessive number of times. Now, an algorithm exists which automatically chooses the right number number of times to run program versions to achieve a given statistically valid result, freeing the compiler writer to concentrate on other things.

Chapter 8

Conclusion

This thesis has developed new ways to remove the human expert from machine learning in compilers experiments. In particular, chapters 5 and 6 present a new method to generate features over the compiler's internal data structures, replacing the haphazard approaches that went before. Chapter 7 offers a novel procedure to manage the data gathering phase so that the researcher no longer need trade off statistical significance for reduced compute time.

This chapter is organised as follows. Section 8.1 summarises the main contributions of the thesis. Section 8.2 critically analyses the work and section 8.3 explores possible future lines of enquiry.

8.1 Contributions

This section summarises the main contributions of the thesis.

8.1.1 Extensible Compiler

Performing machine learning in compilers experiments requires deep access to the internals of the compiler. Most compilers have not, however, been built with the demands of machine learning in mind, and so researchers have resorted to ad hoc solutions, making difficult to maintain changes directly in the source of the compiler.

Chapter 4 presented a suite of software, *libPlugin*, which changes GCC into a research compiler. The compiler is given a set of extension capabilities along the lines of modern plug-in systems. Now researchers can implement their experiments without having to change a single line of GCC's source code and they can share their efforts

in a cooperative, modular fashion. *libPlugin* supports modern software engineering practice and, in addition, comes with a number of predefined tools that make common machine learning tasks trivial.

8.1.2 Feature Generation

Chapter 5 showed that there are an infinite number features that could be chosen for machine learning in compilers. It showed, also, that when left to a human, poor choices would result in the machine learning algorithm being unable to achieve its full potential. The chapter describes a new technique to specify infinite families of features with probabilistic context free grammars that go far beyond the small number of features a human could define. Low level operators needed to search over feature grammars are shown as is a new representation which combines the best aspects of genetic programming with grammatical evolution.

Chapter 6 built upon the feature grammars of the previous chapter, creating a complete system to search for good features from the space defined by a grammar. The quality of features was judged by how well each improved the performance of a machine learning algorithm. The search technique was based on an evolutionary algorithm.

We compared our approach to the prior state of the art using loop unrolling in GCC as an example, target heuristic to replace. We found that our method was able to achieve 76% of the maximum speedup available over 57 benchmarks from the MediaBench, MiBench and UTDSP benchmark suites. This was shown to be far better than the previous state of the art had managed, a feat all the more impressive because no human expert was needed to design and implement the features.

8.1.3 Efficiently Gathering Training Data

In chapter 7 we showed how fixed size sampling plans can lead to either an inadequate sample size to get statistically significant data or to wasted effort. The compiler writer cannot easily predict, ahead of time, how much noise will infect the execution time measurements of each program and so cannot guarantee, with a constant sample size that they will have a large enough sample. To be sure of having good data, he must run programs a very large number of times.

For machine learning in compilers, we are only interested in which optimisations perform best for each example program. Therefore, if we could see after a few execu-

tions of a program version that it had no hope of being the best, then we need not run it further. Ideally we should like to focus all efforts on those program versions which might end up being the best. While some small number of prior works execute a program only until a statistically significant measurement is taken, even these do too much work because they will continue industriously working on program versions which are clearly not going to out do others in the current set.

The chapter developed a system that automatically adapts, managing the iterative compilation so that different program versions are *raced* against one another. As soon as a version falls behind in the race it is removed and no further resources are devoted to it. The remaining program versions continue on in the race until either one wins or the final few are all considered to be equally good.

Taking loop unrolling in GCC and also, separately, the choice of compiler flags, the chapter showed that the racing technique outdid the prior state of the art, needing 77% and 89% fewer executions on average.

8.2 Critical Analysis

The thesis has found a new way to generate features for machine learning in compilers and presented a new algorithm to efficiently manage iterative compilation. This section analyses these new techniques.

8.2.1 Extensible Compiler

GCC is an ageing compiler infrastructure, it is monolithic, cumbersome and awkward. Although the compiler still receives a great deal of development attention and produces competitively optimised code for a wide range of architectures, it is, in this author's view, a legacy compiler written in a legacy language. Newer compilers like LLVM and Open64 are designed more cleanly from the ground up. It may have been wise to have avoided GCC altogether, adding *libPlugin* to one of its successors instead. Additionally, more time should have been spent evangelising *libPlugin*. The software suite has been used by a few researchers only.

8.2.2 Feature Generation

Automatically generating features for machine learning in compilers experiments introduces yet another cost to be accounted for. In the loop unrolling example of the

thesis, each time a feature set was to be evaluated, a prediction was constructed, based on the features. The quality of the prediction was then estimated using previously gathered iterative compilation data (although the final figures for comparison were generated by execution not estimation). Despite using only estimation, the search time for good features was two days or so; had the optimisation under consideration been more complex, making estimation difficult, then the search time could have become excessive. On the other hand, the feature search is trivially parallelisable, which may bring the time to within acceptable parameters. Moreover, while the compiler might be expected to be retuned every time a new version is shipped, it is possible that the features need not be regenerated so frequently.

Prior to the techniques of this thesis, the compiler writer could, at least, be expected to understand the meaning and importance of the features, since he created them himself. Now, however, it is possible that the reasons for improved performance with automatically generated features might be hard to discern; the automatic features are often somewhat impenetrable. If he does not find the performance boost he expects, he must now wonder if this is due to an inadequate feature grammar, in addition to the existing potential causes of insufficient benchmarks, poor machine learning algorithms, bugs in his experimental methodology, etc. These difficulties in understanding the results of machine learning and evolutionary searches have been cited before as reasons to avoid those techniques; typically, as long as the benefit outweighs this price in some field, then those techniques survive.

There are other approaches to learning over tree and graph structures which do not require this type of feature generation. These are tree and graph kernel methods (Aiolli et al., 2007) which operate directly over those data structures. The significant drawback of these methods is that they require the examples to be shipped with the model. Even for our small, loop unrolling example, this would mean including several gigabytes of sata with the compiler; if many heuristics in the compiler were machine learning enabled, the compiler might then comprise terabytes of data. This would be prohibitively impractical, which is why this thesis does not consider it. However, it may be interesting to examine whether kernels could be produced and then streamlined to retain only the most relevant data or if features could be extracted directly from the kernels.

The thesis would have benefited from repeating the experiment with different optimisations and on different platforms.

8.2.3 Efficiently Gathering Training Data

The runtime of each step in the adaptive iterative compilation system of chapter 7 is $\Omega(n^2)$, where n is the number of remaining candidates. For modest experiments this cost is insignificant compared to the large constant in the linear runtime for executing the candidates. However, if massive scale iterative compilation were to be attempted, with tens of millions of candidates, then the cost of the adaptive algorithm itself may exceed the cost of running the candidates.

8.3 Future Work

This thesis has developed novel techniques to remove the need for human involvement when using machine learning for compilers. This section details the possible future work that could be explored going forward.

In the context of feature generation and search, there are several potential avenues for further work. Different feature grammars could be designed for each stage of the compiler when the internal data representations will be different from the register transfer level (RTL) trees that GCC uses for loop unrolling. Feature languages capable of using all possible information whenever a heuristic is produced would mean that all aspects of the compiler would be open to machine learning experiments, not just loop unrolling.

It also may be that different search mechanisms than our evolutionary approach may be even more effective. Perhaps, with enough examples, machine learning might be applied to speed up the search of the feature space.

For the adaptive sampling plans which efficiently manage the iterative compilation during the training data gathering phase there are several possibilities. Different stopping criteria may yield improved results. Similarly, the exact subroutines that compare and remove program versions might be tuned for efficiency. Finally, the initial sample size and the sample size increments might be learned via machine learning to speed up the search.

8.4 Summary

This chapter has reviewed the contributions of the thesis, including a plug-in system to turn GCC into a machine learning capable compiler, methods to automatically search

for good features for machine learning in compilers experiments, and a technique to reduce the cost of generating machine learning training data without sacrificing statistical soundness. The chapter has also critically analysed the work of the thesis and explored the potential for future work.

Appendix A

libPlugin Details

libPlugin is an extensibility library for C which makes providing extension capabilities for applications easy. The library is application agnostic but was specifically constructed to make GCC extensible. With it, the compiler's source code remains clean.

One of the main goals of the project is that absolutely minimal changes should required to GCC to support extensibility. As will be seen, when a fixed heuristic is converted to an extensible one the differences are almost unnoticeable. Indeed, the changes to GCC amount to some ten lines of code in the main function and often only one additional line of code per heuristic. The plug-in system is extremely simple to use without compromising power and flexibility.

There are two primary objects types in *libPlugin*; plug-ins and extension points.

A.0.1 Plug-ins and Extension Points

Plug-ins are modules encapsulating a areas of functionality for the compiler. Plug-ins provide services to each other through extension points; plug-ins use services of other plug-ins by extending those points. Moreover, only plugins can provide extension-points so that even core application services are bundled together into plug-ins, even though they might always be present and not optional.

In GCC, for example, there is a core plug-in which allows loop unrolling factors to be overridden on a perloop basis. This plug-in offers an extension point called `gcc-rtl-unroll-and-peel-loops.override` [1]. If a developer wants to force cer-

[1] GCC unrolls loops at different stages in the compilation process. The first but simplest is performed at a high-level while the code is still in an AST form (called GIMPLE). The more powerful optimisation operates on loops which have been converted to a lower level form, the Register Transfer Level (RTL). That optimisation performs both unrolling and peeling at once. Thus, the extension-point mentioned above is for the RTL level unrolling optimisation. A similar plug-in enables overriding for the GIMPLE

tain unroll factors for particular loops, they can write another plugin which extends the point, `gcc-rtl-unroll-and-peel-loops.override`.

A.0.2 Plug-in File Format

For each plug-in there is an XML description file. This file tells *libPlugin* everything it needs to know about the plug-in from its dependencies to the extension-points it provides and uses. Some plug-ins need no more than this description file, while others might need some C code to drive their functionality. In the latter case, the plug-in has some number of shared libraries in addition to the XML description.

The minimal plug-in specification is shown below. The plug-in will cause GCC to print `Hello, World!` to the standard output. This XML plug-in specification is either placed on a special plug-in search path or mentioned on the command line to GCC.

```
1 <?gcc version="4.3"?>
2 <plugin id="hello-world">
3     <extension point="message.start">Hello, World!</extension>
4 </plugin>
```

Each plug-in specification file must contain valid XML indicate what applications it should work with and contain a valid `<plugin/>` element. The simple file above is described, line-by-line, below.

In line 1, the plug-in declares that it applies to GCC with version at least 4.3 [2].

In line 2, the plug-in gives itself an identifier. Plug-ins use identifiers to declare that they depend upon each other and users also give these identifiers to load optional plug-ins from the command line. Plug-ins can, in fact, be anonymous, but it is considered good practice to always name them.

Line 3 says that this plug-in extends another plug-ins extension point. The identifier of that point is `message.start`. *libPlugin* will ensure that the plug-in providing that extension point is loaded and that only one such plug-in exists. In this case, the extension point is simple, and happens to be provided by the `message` plug-in. It will

level loops.

[2] *libPlugin* is a library that can help to make any application extensible, not just GCC. Some plug-ins may work with more than application and can provide multiple directives to that effect - indeed some plug-ins work with all applications and can give a catch all directive of `<?plugin version="1.0"?>`. Each application will decide whether the version string is suitable for it and *libPlugin* will ignore any unsuitable plug-ins. This simple multi-application will, in the next version, allow plug-ins to target the linker, assembler or driver as well as different language specific parts of GCC. These capabilities, however, are not of particular interest to machine learning tasks and so will not be elaborated upon further.

print the text contents of the extension to the standard output[3].

Line 4 ends the specification.

A.0.3 Plug-in Selections and Dependencies

Not all plug-ins are loaded by the system; the plug-in search path may include many plug-ins and several will be required only on some occasions. For example, there is a plug-in provided by default which will print a list of all passes each function goes through as GCC compiles it. That plug-in is useful for debugging purposes but would not be interesting during the majority of compilations. Conversely, if a developer has installed a plug-in which implements a useful optimisation, he may want a plug-in to be always present.

A.0.3.1 Lazy and Eager Plug-ins

In *libPlugin* there are several ways to indicate which plug-ins should be loaded during a compilation. The first is the distinction between *eager* and *lazy* plug-ins. Lazy plug-ins are not loaded unless they are required by some other mechanism. Eager plug-ins are always loaded.

A plug-in is marked as lazy by adding attribute `lazy="true"` to the plug-in XML specification file. All plug-ins without this are eager.

```
1 <plugin id="..." lazy="true">
2     ...
3 </plugin>
```

A.0.3.2 Loading From Command Line

Users may inform *libPlugin* that they require certain plug-ins to be loaded via command line arguments. *libPlugin* will check that the plug-ins exist on the path and if they are marked as lazy plug-ins, then it will put them in the required set and cause them to be loaded.

To load a lazy plug-in from the command line, the user gives a comma separated list of plug-in identifiers. For example, with the command line below, GCC will log all

[3]The `message` plug-in is quite powerful. It can send formatted text to different files and even sockets at the beginning and end of the application as well as in response to various events. This makes is very simple, for example, to log how all of the loops in benchmark were unrolled.

passes each function goes through during compilation and will additionally print how the default unrolling heuristic would optimise each loop in the file `foo.c`[4].

```
gcc -plugins gcc-print-pases,gcc-print-unrollable-loops foo.c
```

A.0.3.3 Recursive Dependencies

Plug-ins need to depend upon the services provided by other plug-ins. *libPlugin* provides several ways to specify these dependencies.

The simplest dependency is implicit. When a plug-in extends an extension point, the owner of the extension point will be loaded (if not already). A plug-in can achieve the same effect programmatically in its life-cycle methods by simply getting the address of the extension point.

The second method is explicit. The plug-in'XML specification can contain a `<requires plugin="foo"/>` to require plug-in `foo`. [5] This can also be achieved programmatically.

A.0.4 Plug-in Life-Cycle

Plug-ins have a defined life-cycle model which allows them to perform suitable initialisation and clean up tasks in the knowledge that services they require from other plug-ins will be operational.

The *libPlugin* plug-in manager goes through a number of phases. An example of the movement through these phases is given in figures A.1 and A.2.

A.0.4.1 Parsing

The first phase for the plug-in manager is to parse all of the plug-in XML specification files it can find in the plug-in search path. *libPlugin* will know which plug-ins exist and are eager. Additionally, it determines which extension points are provided by which plug-in.

[4]Both of the plug-ins, `gcc-print-pases` and `gcc-print-unrollable-loops` are provided by default in the GCC implementation of *libPlugin*

[5]The `<requires/>` element also allows a specific range of plug-in versions to be given.

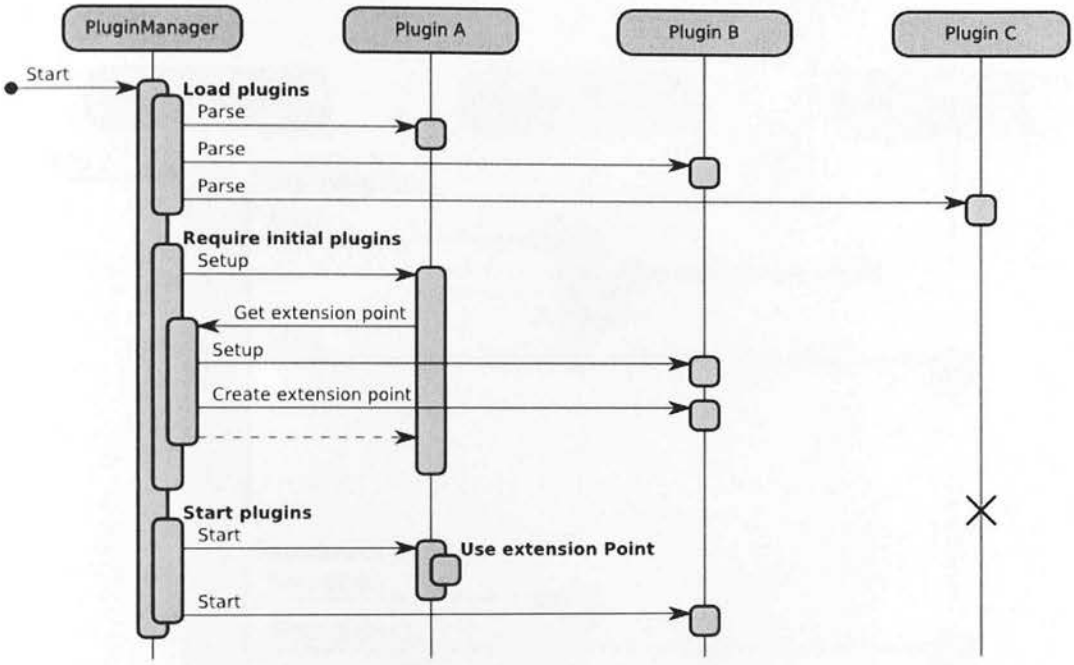


Figure A.1: Plug-in start sequence in UML form.

First the XML plug-in specification documents for each plug-in on the search path are parsed.

Next any plug-in that is not marked as `lazy` or is required by the user's command line is marked as required. (In this example, only Plug-in A meets this criterion).

Required plug-ins then have their `setup` life-cycle method called. This method may cause other plug-ins to become required by, in this case, programmatically asking for the extension point of another plug-in, which is here provided by plug-in B and thus plug-in B subsequently becomes required. Plug-ins cannot use other methods extension points during this method (because the other plug-in may not be set up).

Once all required plug-ins have been set up, their `start` life-cycle method is called. Plug-in A can now use the extension point from plug-in B.

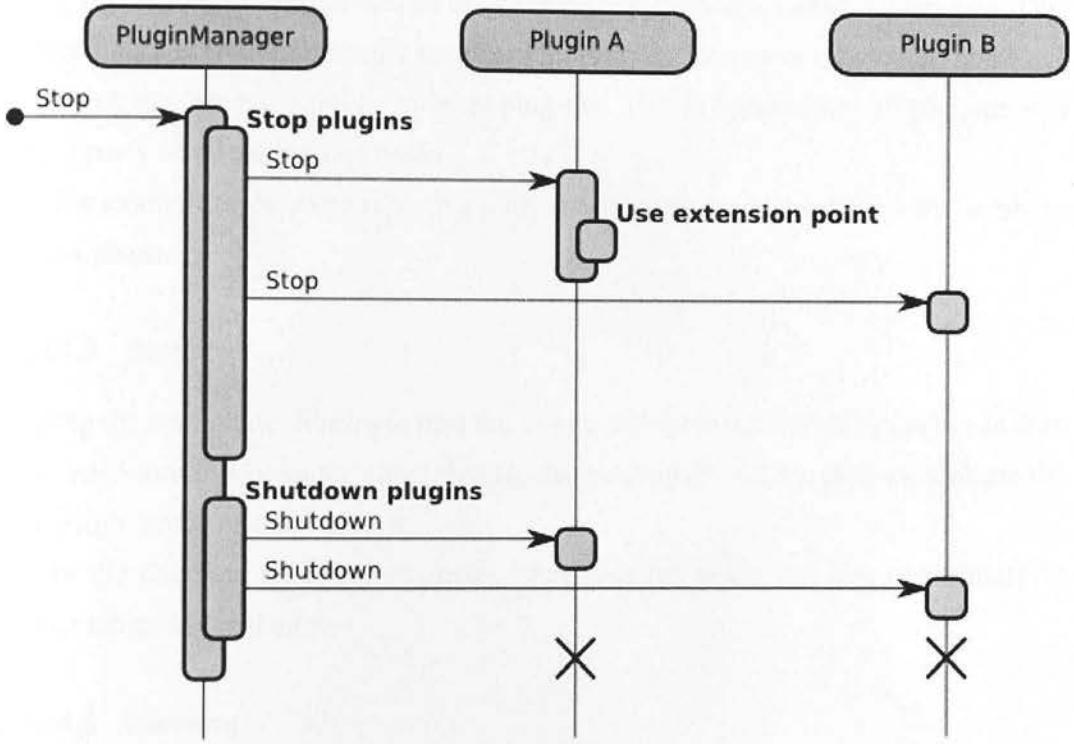


Figure A.2: Plug-in stop sequence in UML form.

All live plug-ins have their `stop` life-cycle method called. During this phase, plug-ins know that other plug-ins are still active and can continue to use their extension points. This allows plug-ins to, for example, send statistics to a database plug-in, knowing that the database connection is still open.

After all plug-ins have stopped, their `shutdown` life-cycle methods are called. Plug-ins know that no other plug-ins will use their extension points so they can close any resources they might still have open.

A.0.4.2 Setup

During the setup phase, *libPlugin* will determine which plug-ins need to be required. It starts with the initial set of eager plug-ins and those which are specified on the command line. Then it require plug-ins from dependencies in the specification files of already required plug-ins.

As each plug-in is processed its life-cycle `setup` method is called if it has one. This method might programmatically require other plug-ins. It may not, however, make use of any of the extension points of other plug-ins. This is because not all plug-ins will be properly setup during this phase.

For example, a database reporting plug-in might open a connection to the database in this phase.

A.0.4.3 Start

During the start phase, *libPlugin* runs the `start` life-cycle method of each plug-in than has one. From this point on, since all plug-ins are properly set up, plug-ins can use the extension points of other plug-ins.

In the database reporting example, other plug-ins might use this opportunity to create tables in the database.

A.0.4.4 Running

The system is initialised and all plug-ins have started. GCC then compiles the source files it has been asked to compile. Depending on the plug-ins that have been loaded, some of GCC's default behaviour will have been altered by the loaded plug-ins.

A.0.4.5 Stop

Once GCC has compiled the source files it was given, it stops loaded plug-ins. Every plug-in with a `stop` method will have that method called. Plug-ins may still use the extension points of other plug-ins and should keep their own usable.

If one plug-in, for example, has been collecting statistics during the running of GCC then it could now send them to a database plug-in, knowing that its extension points are still active.

A.0.4.6 Shutdown

Finally, before releasing all resources, *libPlugin* will call the `shutdown` life-cycle methods of any plug-ins that have them. This gives those plug-ins the opportunity to close any resources of their own which *libPlugin* cannot know about.

A.0.4.7 Life-cycle Methods

During the different phases, plug-ins might need to specify functions to be run. The plug-in XML specification format allows plug-ins to indicate which methods those are, if any. The plug-in attaches a shared library with the `<library>` element. Within that element the plug-in may use `<setup>`, `<start>`, `<stop>` or `<shutdown>` elements to show that it has life-cycle methods in the library.

Below, a simple plug-in will print a message during the start phase. First is the XML specification.

```
<plugin id="foo">
    <library path="foo.so">
        <start/>
    </library>
</plugin>
```

Next is some suitable code which should be compiled into library `foo.so`. [6] The function will print the identifier of the plug-in at start up.[7]

```
#include <stdio.h>
#include "libplugin/Plugin.h"

bool Plugin_start( Plugin* plugin ) {
    printf( "Plugin, %s, was started!\n", Plugin_getId( plugin ));
    return TRUE;
}
```

A.0.5 Extension Points and Extensions

Plug-ins bundle together related functionality into sensible units. The majority of the work, however, is performed at a finer grained level, that of extension points. At it's most simple an extension point is simply a named object which has an `extend` function with which other plug-ins can pass it snippets of XML. The XML can be arbitrarily

[6] The predefined name `Plugin_start` can be changed by including a `symbol` attribute in the `<start>` element.

[7] There already exists a plug-in for printing formatted messages to various outputs.

complicated and include pointers to symbols from from the extending plug-in's shared libraries, so there is no limit to the power of the extension mechanism.

For example, GCC provides an extension point, `gcc-rtl-unroll-and-peel-loops.overri` which allows other plug-ins to override the default loop unrolling heuristic for any loops it chooses. If a plug-in includes the snippet below in its specification file, it will extend that extension point, asking for loop two in function `bar` from file `foo.c` to be unrolled 10 times.[8]

```
<extension point="gcc-rtl-unroll-and-peel-loops.override">
    <loop
        main-input-file="foo.c" function="bar" loop="2"
        times="10"/>
</extension>
```

The implementation of the extension point itself is in two parts. First there must be some C function to accept any extensions. In pseudo-code it looks something like this:

```
list overrides = NULL;

bool overrideExtend(
    ExtensionPoint* self,
    Plugin* extender,
    xmlNodePtr specification
) {
    replace unroll heuristic with overrideUnroll;
    for each child in specification {
        append child to overrides;
    }
    return TRUE;
}

int overrideUnroll( loop* lp ) {
    spec = first element in overrides matching lp;
    if( spec == NULL ) return previous heuristic;
    else return spec.times;
}
```

The extension function remembers what overrides it is given. It also needs to replace the default unroll heuristic with something that will use the overrides when given. In fact the unrolling heuristic is represented by another extension point so it can be programmatically overridden.

This demonstrates one of the powerful aspects of the extension point system; the

[8]The unrolling override extension point provides more functionality than this.

ability to compose different extension points to give layers of functionality. The first extension point allows low-level alterations to the the heuristic and another gives a high-level but less capable wrapper.

The plug-in must also declare the `gcc-rtl-unroll-and-peel-loops.override` extension point. It does this by putting the following in its XML specification:

```
<extension-point id="gcc-rtl-unroll-and-peel-loops.override">
    <extend symbol="overrideExtend"/>
</extension-point>
```

This declaration informs *libPlugin* that whenever another plug-in extends the extension point the `overrideExtend` function should be called.[9]

Although the extension mechanism is very simple to arrange it does require some coding of the extension function which invariably involves an amount of tedious XML processing[10]. Since one of *libPlugin*'s goals is to make extensibility as simple as possible, it offers are number of short cuts for defining powerful extensions for the most common cases with almost no code. The following sections describe the easy ways to create and use convenience extension points.

A.0.5.1 Events

Events are a common programming pattern that are extremely useful in an extensible compiler. Consider, for example, the case when a user would like to know what loops have been unrolled and with what unroll factor. They might choose to log this information to a file or to a database or to aggregate it in some other way as the loops are unrolled. If the compiler fires an event every time it unrolls a loop, then users can listen to those events and do anything they want. The compiler is thereafter free from worrying about whether is has supported every possible user interaction. *libPlugin* makes creating, firing and listening to events trivial. This section describes how that is done.

Event extension points allow one plug-in to publish events to other plug-ins. Plug-ins register their interest in listening to the event by extending the event extension point. Each extending plug-in will give a call back function to be invoked when the event occurs. The owning plug-in will give the system a function pointer (also of the same type) which will be replaced if any other plug- in extends this event. The

[9] In the actual plug-in the `overrideExtend` function is provided in a separate, optional shared library. There are other small differences in the real code which take advantage of more advanced *libPlugin* features.
It is also possible to create extension points by pointing to a factory method or programmatically in the `setup` life-cycle method of a plug-in.

[10] *libPlugin* is built on top of the open source `libXML2` library.

replacement will call each of the call back functions from the listeners.

The process is best shown by example. Suppose that one plug-in would like to report an event called `something_happened` and parametrise this with a number and a string. First we see how the plug-in must write its C code to declare and to fire the event.

```
// Declare an empty function that is the same as firing
// the event to no listeners
void something_happened_empty(int number, char* string) {}

// Declare a function pointer for the event
void (*something_happened)(int number, char* string) =
    something_happend_empty;

// Later in the code, fire the event
something_happened(100, "it happened!");
```

The event is nothing more than a function pointer with the right prototype. The plug-in can call it whenever it needs to. Initially, the function it points to does nothing. If any other plug-in is listening to the event then *libPlugin* will have replaced the function pointer with a new function which informs all listeners of the event. The changes to the code are kept to the bare minimum. [11]

To tell *libPlugin* about the event, the plug-in adds this to its XML specification:

```
<event id="app.core.something-happened"
    signature="void f( int, char* )">
    <call symbol="something_happened"/>
</event>
```

The XML gives the event an identifier and tells *libPlugin* the name of the function pointer to replace and what the signature of the event is. We will see why the signature is necessary later.

Listening to the event is just as straightforward. The listening function is written with the same prototype as the event:

```
void handle_something_happened(int number, char* string) {
    printf("It happened! number=%d string='%s'\n", number, string);
}
```

And the listener function is declared to the system: [12]

[11] Often the function pointer will be initialised to `NULL` and a null check will be made before firing the event. This is more efficient and avoids computing needless arguments.

[12] Factory methods can also be used to create events and event handlers; both are objects, not just functions and the example declarations here show only the most convenient usage.

```
<extension point="app.core.something-happened">
    <callback symbol="handle_something_happened"/>
</extension>
```

Now whenever the event is fired the call back is triggered.

A.0.5.1.1 Event Handler Creators *libPlugin* goes much further. Plug-ins can provide event handling services to other plug-ins which often means that powerful effects can be achieved without writing any C code at all.

For example the `message` plug-in provides an event handling service which plug-ins can use to create an event handler which logs event information when the event is fired. The extension below has exactly the same effect as the previous, C-based one but requires no shared library. [13]

```
<extension point="app.core.something-happened"
           create="message.event-logger">
    <text>It happened! number=</text>
    <arg-print index="0" format="%d"/>
    <text> string='</text>
    <arg-print index="1" format="%s"/>
    <text>'</text>
    <br/>
</extension>
```

A.0.5.1.2 Dynamic Code Generation There has been a certain whiff of smoke and mirrors in the description of the working of the event extension points. In particular the issue of how *libPlugin* replaces the event function pointer so that it points to new function which will call the waiting listeners has been glossed over. In fact, this is strictly not possible using C. The listeners of two different events may have different signatures and one function cannot call both functions types with having them hard wired into it, so the 'call all listeners' function is impossible to write. Moreover, there is no function prototype suitable for that function which would allow it to be put at all the function pointers of the events. Not only can we not write the function in C, but we could not use it even then.[14]

Indeed the situation is worse because the actual function prototypes have to be changed; *libPlugin* is really an object-orientated system in C that allows users to stick

[13] There is coming a JavaScript plug-in which allows other plug-ins to run arbitrary scripts to be run in response to events.

[14] C's variable length arguments are not sufficient to solve this problem.

with simple C prototypes whenever convenient and auto constructs the objects for them. This requires prototypes to be created that have a `this` pointer to the object and then the remainder of the original arguments. Again, due to C's lack of reflection this is not possible in C alone.

This difficulty is overcome by dynamically generating small thunk functions thunk that marshal arguments into and out of reflective arrays. When an event is listened to, *libPlugin* creates a function which has the same signature as the event. This function gathers the arguments into an array and passes it to a generic event dispatching function. That function in turn will call each of the listeners' call back functions which will generally require the arguments to unmarshalled again into the normal native argument type using another dynamically generated thunk.[15]

These efforts allow the user to make simple event handlers easily and naturally but also allows powerful generic event handlers can also be created. *libPlugin* takes care of all of the hard work.

A.0.5.2 Around Advice

One of the primary needs of machine learning in compilers is to be able to replace the default behaviour of an heuristic. To support this, *libPlugin* borrows a concept from aspect orientated programming (AOP). AOP allows developers to add *advice* to methods that have already been written. One form of this advice replaces the method with a new one which receives the same original arguments. The advice can perform any operation it desires but in particular can also, if it needs to, call the original method it replaced. In fact, these advices can be layered, with one method being advised multiple times, and each layer of advice can call the next one down if it wants to. The AOP formulation has been very successful and has been demonstrated in a large number of real world projects; it is also a perfect fit for the machine learning in compilers requirements.

libPlugin allows plug-in writers to specify that a function can be advised and for plug-ins to advise that function. The formulation is very similar to that of events with a few small changes; the functions involved may now have return types; the XML

[15] In fact the call back function of all event handlers accepts the reflective array version of the arguments together with a `this` argument. This allows generic event handlers such as the message event logger from the previous section to work. If the user provides a non-reflective function - which *libPlugin* allows believing that convenience for the user should be paramount - a thunk is dynamically created to unmarshall the arguments back into the native form and that thunk becomes the event handler's call back.

description of the advisable function uses an `<around>` tag rather than the `<event>` tag and there are additional API functions for advice to get a pointer to and call the next advice in the stack. Additionally, around advice can be built generically in the same way that event handlers can so that plug-ins do not always have to resort to C code to advise methods. For all of this to work, around advice requires the same kind of dynamic code generation as events.

Typically, around extension points do not exist in isolation. There is a more powerful concept called a *join point* which is more useful for defining replaceable heuristics. Join points contain an around extension point (and two events) within them and there are no practical advantages to using an around extension point by itself. Join points are discussed next.

A.0.5.2.1 API to Call Through the Advice Stack The simplest way to advice a function is to write another function with the same prototype. When this advice is on the top of the advice stack (or called by higher level advice) it will completely replace the function it is advising.

So, if we had an original function,

```
int heuristic(int number, char* string) {
    return number * strlen(string);
}
```

We could completely replace it with a new function which did the same thing but added one to the answer.

```
int replacement_heuristic(int number, char* string) {
    return number * strlen(string) + 1;
}
```

Being unable to reuse the previous function (or rather the next advice down on the stack which might be just the original heuristic) is extremely annoying. *libPlugin* allows advice easy access to the next advice on the stack but only if they use one of the object-oriented forms of advice (recall that events were really object oriented to, they just appear to be functions because *libPlugin* generates convenience object for the user - the user could also have created the object himself).

To use the object oriented version there must be a `self` pointer as the first argument.

```
int replacement_heuristic(
    AroundAdvice* self,
    int number, char* string
) {
    return number * strlen(string) + 1;
}
```

We must also inform *libPlugin* in the plug-in XML that the advice function has this pointer as first argument:

```
<extension point="heuristic">
    <callback symbol="replacement_heuristic" type="non-static"/>
</extension>
```

16

From this `self` pointer we can get information about parameter types, extension and plug-in identifiers and much more. However, at present we are interested getting a function pointer we can call for the next advice on the stack. This we can do with the function `AroundAdvice_getCallNextFn`; it returns an untyped function pointer so we have to cast it to the proper type: [17]

```
int (*next)(int, char*);
next = (void (*)(int, char*))AroundAdvice_getCallNextFn(self);
```

The `next` function can then be called as normal. If the current advice is the only one on the advice stack then the next function will call the original function; otherwise it will call the next advice which can, should it need to, call its next advice and so on down to the original.

To add one to the original heuristic the code would then be:

---

[16] *libPlugin* allows three convenience methods for automatically constructing the advice object from a single call back function. The default is without the `self` argument; in addition to the simple one with the `self` argument there is one which takes a `self` argument and a reflective array of the remaining arguments. This latter form is useful for generic advice.
The user could alternatively give a symbol which points to an advice object or to a function which creates advice objects. These forms give the user complete power at the expense of having to slightly write more code.
All of these conveniences are provided to events as well and at many other places in *libPlugin*, making it a very easy system to use and requiring as few lines of code as possible.

[17] This function will create the required dynamic thunk code if necessary and is cached thereafter. There are other functions to get the `next` function in different formats.
It is also possible for an advice object to demand to be the top advice on the stack. This is achieved by adding an attribute `selfish="true"` to the extension specification, but like most things in *libPlugin* can also be done programmatically. If any other advice tries to advise the function after a selfish advice has been applied an error is generated and the program stops.

```
int replacement_heuristic(
    AroundAdvice* self,
    int number, char* string
) {
    int (*next)(int, char*);
    next = (void (*)(int, char*))AroundAdvice_getCallNextFn(self);
    return next(number,string) + 1;
}
```

A.0.5.3 Join-Point

Around extension points allow functions and heuristics to be modified. They have some small practicality limitations, however. A typical usage pattern for altering heuristics is that we may wish to receive an event when the function is first called with the arguments passed to it and another when the function terminates, this time with the return value as well as the arguments. These events are useful when some kind of reporting is required which does not override the behaviour of the advised function. We cannot simply have some around advice which performs the logging and then delegates to the next advice on the stack without altering arguments or return value since we cannot guarantee the ordering on the advice stack.

AOP solves this problem with a concept called a *join point* and *libPlugin* borrows that concept. A join point consists of exactly the two event extension points and an around extension point that we need as shown in figure 4.11. When a join point is called, first all of the before event listeners are notified with the function's parameters. Then the top advice on the around advice stack is called with those parameters, which may or may not call further down the advice stack. Finally the after event listeners are notified with the return value from the top advice and the original parameters of the function call.

Creating and using a join point in *libPlugin* is very simple. For example, if the C code for the loop-unrolling heuristic is originally as our initial example:

```
int decideUnrollTimes(loop* lp) {
    int times = /*the heuristic*/;
    return times;
}
```

Then we can turn it into a extendible join point by converting it to function pointer instead. This requires only one additional line of code and all uses of the function remain exactly as they were; the compiler is not cluttered with ugly extensibility code.

The heuristic now looks like this:

```
static int decideUnrollTimes_original(loop* lp) {
    int times = /*the heuristic*/;
    return times;
}
int (*decideUnrollTimes)(loop* lp) = decideUnrollTimes_original;
```

The join point needs to be declared to *libPlugin* in a plug-in XML specification file, giving an identifier for the join point, the prototype of the function so that dynamic code can built for it and the function pointer to be replaced: [18]

```
<join-point id="decide-unroll-times"
    signature="int f(loop*)">
    <call symbol="decideUnrollTimes"/>
</join-point>
```

The join point is not in itself an extension point. Instead, a join point creates three extension points. If the identifier of the join point is `x`, then the first event extension point will have identifier, `x.before`, the around extension point will have `x.around` and the last event will have `x.after`. These extension points are then used as normal.

If no plug-in listens to one of the join point's events or places advice on the around stack then the function pointer, `decideUnrollTimes`, will still point to its original value, the function `decideUnrollTimes_original`. Only if necessary is any dynamic code constructed and the function pointer updated. In this way, just as for events and around extension points, there is practically no cost to making the compiler extensible.

A.0.5.4 List

The simplest type of convenience extension point provided by the system is a list of values. The null terminated list can contain a pointer to any type of data and other plug-ins can append values to the list.

For primitive types (integers, floating point numbers and strings) the element to be appended can be given directly in the extending plug-in's XML description. More complex types are handles by either simply providing a symbol which points to the element in a shared library or by giving a factory

[18] As with most things in *libPlugin*, this is just one of the convenient methods. There are other ways to achieve the same result which are useful if more power is required.

A.0.5.5 Hook

A hook allows a function's implementation in one plug-in to be replaced by another plug-in. A hook in the owning plug-in is simply a function pointer which another plug-in can overwrite. The owning plug-in can then call the hook whenever it needs to. Only one plug-in can extend a hook; the system will report an error if two plug-ins attempt to extend the same hook.

Hooks are extremely limited; for nearly all cases the join-point extension (see above), which is much more powerful should be used instead. However, there is a cost to join-points because no dynamic binding code needs to be generated. Hooks, on the other hand, provide extensibility at only the cost of an indirect function call. They should be used only when the hook will be called so often that the greatest efficiency is required.

All of these convenient methods for creating extension points means that the compiler can be 'marked up' for extensibility with a tiny amount of code which is hardly noticeable. The original purpose of the code remains uncluttered by extraneous additions just for the sake of extensibility.

Moreover, *libPlugin* takes great pains to ensure that extensibility is as efficient as possible, in particular in the case where the user does not wish to extend some part of the compiler. Consider, by comparison the situation in the original example where the heuristic was littered with extensibility code. This code tests whether each individual extension is required and these tests must be undertaken regardless of whether the user has requested the extension. Naïve implementations using shared libraries can look marginally cleaner but still scour through lists at each invocation, only to find that no extension for the particular point has been made. No extensibility library for C can match *libPlugin* for it's power, simplicity, convenience and efficiency.

A.0.6 Machine Learning Plug-ins

libPlugin for GCC comes with a number of plug-ins that are useful for machine learning tasks. The following sections give a brief introduction to each.

A.0.6.1 Perfmon

The `gcc-perfmon` plug-in allows GCC to instrument functions it is compiling to count the number of cycles spent in each function. Cycle counts are very useful when performing iterative compilation on functions or smaller code elements as they give timings to very high precision when other techniques might be too coarse grained to time element. There can be a downside that occasional context switches will cause outliers in the data, however, these outliers are usually so far out that they are easily purged from the data.

When instrumenting a function `gcc-perfmon` will transform it from this:

```
fn() {
    fn_body;
}
```

Into something like:

```
static callcount_t fn_callCount = 0
static cyclecount_t fn_cycles = 0
fn() {
    fn_callCount++;
    cyclecount_t t0 = CYCLECOUNT();
    try {
        fn_body;
    } finally {
        cyclecount_t t1 = CYCLECOUNT();
        fn_cycles += t1 - t0;
    }
}
```

[19]

The cycle count for a function can also be paused when another function is called. By default the count is always paused when another function is called, but for some functions that may be inappropriate. Which functions to pause for can be completely under user control.

Code is also inserted into the current compilation unit to dump the gathered statistics when the program exits.[20] The statistics can be output to a file as XML, SQL or JSON. In all those cases an identifier for the current compilation can be added which is necessary when doing iterative compilation.

Additionally, the JSON format is compatible with CouchDB and open-source,

[19] In fact the code is slightly different for efficiency reasons and to support callee pausing. Also, the variable names are compiler temporaries that cannot clash with any names in the source code.

[20] This is done by adding a compilation destructor function.

loosely structured, document orientated database. These JSON data can be sent automatically to the database, meaning that every time the program is run the database is informed of that without further effort from any party. When sending to the database additional information is supplied such as the identity of the machine that program was run, the user name, dates and times, etc. The destination for the statistics is initially just the standard error stream, but this can be changed by a plug-in at compile time or even at run time by environment variables. [21]

The plug-in is a lazy plug-in, meaning that it must be explicitly invoked to be loaded by the system. The plug-in can be invoked by adding to the command line, `-plugins gcc-perfmon`, or by another plug-in extending one of its extension points.[22]

There are two join points in the plug-in, `gcc-perfmon.should-instrument-current-funct` and `gcc-perfmon.should-pause-callee`, and an extension point, `gcc-perfmon.settings`. The latter extension point allows a simple specification of which functions to instrument and pause for, while the join points give programmatic control over the same.

A.0.6.1.1 `gcc-perfmon.should-instrument-current-function` This join point takes no arguments and returns true if the current function should be instrumented for cycle counting and false otherwise. By default, all functions are instrumented. Any plug-in wishing to alter this can advise this join point and look at GCC's `current_function` pointer to see what is being compiled.

A.0.6.1.2 `gcc-perfmon.should-pause-callee` To find out if a call should pause the cycle count, this join point takes the name of the function being called and a pointer to the AST object for the call. The join point returns true if the call should pause and false otherwise. By default is always returns true.

A.0.6.1.3 `gcc-perfmon.settings` The `settings` extension allows a very simple specification of which functions to instrument and which to pause for. The user gives wild-carded[23] lists of functions to instrument or to pause for. Each element of the lists either includes or excludes the functions it names and the elements processed in document order. A typical example might look like:

[21] At compile time it can also be set on the command line or through environment variables. This is because *libPlugin* also supports full variable expansion mechanisms from those sources (and from inside plug-in specifications). Variable expansion is very useful for writing concise plug-ins but does not particularly aid machine learning goals, so is not discussed in this thesis.

[22] There are several other ways to cause a plug-in to be loaded, it as described in previous sections.

[23] The wild cards used are POSIX glob patterns

```
<?gcc version="4.3"?>

<plugin id="perfmon-example">
    <!-- Extend the settings and dump to file
     output.json -->
    <extension point="gcc-perfmon.settings" file="output.json">
        <!-- Say what to instrument -->
        <instrument>
            <!-- Initially, everything is instrumented, but
              if we only want to instrument a few functions,
              then we start by excluding everything -->
            <exclude main-input-file="*" function="*"/>
            <!-- Now list what to include, two from foo.c
              and all in bar.c -->
            <include main-input-file="foo.c" function="foo"/>
            <include main-input-file="foo.c" function="bar"/>
            <include main-input-file="bar.c" function="*"/>
        </instrument>
        <!-- Say what calls to pause on -->
        <pause>
            <!-- Initially, everything is paused -->
            <!-- We will not pause for transcendentals -->
            <exclude function="sin"/>
            <exclude function="cos"/>
            <exclude function="tan"/>
        </pause>
    </extension>
</plugin>
```

[24]

The extension point itself is built on top of the two join points from the plug-in and, to some extent, demonstrates how easy adding extensibility is with *libPlugin*.

A.0.6.2 Trace

The `gcc-trace` causes GCC to instrument the files it is compiling to create traces of each basic block as it is executed. These traces create very large data files but may

[24] Match attributes where the pattern is a single * can be left out

offer interesting possibilities for machine learning.[25]

The plug-in instruments the code so that a print statement is placed at the beginning of each basic block. The print statement is directed to a file and spits out only an identifier number for the basic block followed by a new line. The file is the standard error stream by default but can be overridden by a plug-in or even at run time through environment variables. The trace files are normally so big that it is recommended to pipe them to a consuming program rather than store them directly.

Additionally, during the compilation the mapping of identifiers to basic blocks is written out to a file so that traces can be reverse engineered to find source file, function and line numbers for the basic blocks in the trace.

The plug-in allows the specification of which functions to trace in a very similar way to the methods for `gcc-perfmon` and so that is not further discussed here.

A.0.6.3 Command Line

Iterative compilation at the whole program level often involves simply searching through different command line options to find which produce the fastest program. In other cases, some benchmarks set various command line arguments which might conflict with the desired experiment, such as turning off loop unrolling when the experiment needs to investigate different unrolling strategies.

Although this changing the GCC command line is conceptually simple, the benchmarks used are not often compiled with a direct shell command to GCC. Instead, there is usually a make file involved and sometimes this can be almost impenetrable. Deconstructing the make file and re-engineering it if necessary is both time consuming and error prone. *libPlugin* solves this by allowing command line arguments to be altered from a plug-in. [26]

The plug-in is simply called `command-line` and is available to all *libPlugin* enabled applications, not just GCC. It contains only one extension point, `command-line.modify`.

A.0.6.3.1 `command-line.modify` This extension point allows command line arguments to be removed and inserted before the program is run. [27]

[25] Some on going work is looking at how different input data sets can effect program performance and we hope that differences between execution traces may act as a proxy for features of the data.

[26] To be loaded a plug-in need only be eager and on the search path. That path can be set by environment variable. This makes command line modification easy and certain.

[27] That is to say, before the program leaves the start phase of the *libPlugin* life-cycle model.

Arguments are removed in sequences matching a wild card pattern. While one often wants to remove only a single argument at a time, there are situations when one argument indicates that next is a parameter for the first argument. Typically both must be removed at once. The following example removes all single arguments beginning with `-O` and double arguments where the first argument is `-I`.

```
<extension point="command-line.modify">
    <remove><arg>-O*</arg></remove>
    <remove><arg>-I</arg><arg>*<arg></remove>
</extension>
```

Arguments can be inserted at the front of the argument list.[28] Argument sequences can be added in the same fashion as they are removed. The example below inserts `-O3` at the front of the command line.

```
<extension point="command-line.modify">
    <insert><arg>-O3</arg></insert>
</extension>
```

The insertions and removals are processed in document order. A useful pattern for iterative compilation, therefore, is to first remove all of the arguments that will be iterated over and then insert the ones from the current search point.

A.0.6.4 Loop Unrolling

Loop unrolling is a well studied optimisation which nevertheless still has room for improvement. It is commonly targeted for machine learning experiments since researchers feel that if gains can be won against such a mature optimisation then they truly demonstrate the validity of their techniques.

In GCC there are two places where loop unrolling is performed. The first is in the high level AST, while the second is much later after the source has been lowered to RTL. The second is much more capable and has been around for much longer. It not only performs several flavours of unrolling but also manages loop peeling at the same time.

libPlugin offers a full featured plug-in to interact with the more powerful RTL unrolling optimisation. A similar plug-in is in development for the AST level code. This section discusses the former only in a plug-in called `gcc-rtl-unroll-and-peel-loops`. The plug-in provides several services related to printing unrolling information about

[28] This extension point does support insertion elsewhere, but plug-ins can programmatically alter the command line should they need more control

loops and querying unrolling status and legal values, however, the focus here will only be on how to force the unrolling and peeling decision for individual loops.

The unrolling plug-in has one join point to programmatically control the unrolling decision and one extension point, built on top of that join point, that allows a simple, no C code specification to be given. The join point is called `gcc-rtl-unroll-and-peel-loops.decis` and the extension is `gcc-rtl-unroll-and-peel-loops.override`. Only the `override` extension point is described here.

A.0.6.4.1 gcc-rtl-unroll-and-peel-loops.override This extension point allows the user to specify which loops should be unrolled and by how much. [29]

Extensions give a list of wild carded `<loop>` elements which match source file names, function names and loop numbers. Each of these `<loops>` gives the number of times the loop should be unrolled (or may say to use the default). [30] The elements are processed in document order until a match is found and that then gives the unroll factor for the loop. If no match is found the default heuristic is used. This ordered processing means that the most specific matching patterns should appear first.

For example, the plug-in specification below unrolls loop two in function `foo` ten times, all other loops in that function by the default heuristic and any loop any other function will not be unrolled.

```
1 <extension point="gcc-rtl-unroll-and-peel-loops.override">
2     <loop main-input-file="foo.c" function="foo" number="2" times="10"/>
3     <loop main-input-file="foo.c" function="foo" number="*" times="default"/>
4     <loop main-input-file="*" function="*" number="*" times="0"/>
5 </extension>
```

[31]

There is also a plug-in, `gcc-print-rtl-unroll-and-peel-loops` which will log a list of all the loops in each function. Information about whether the loop can be unrolled or peeled, and if so in what flavours and by how much may be included as well as what unrolling or peeled was actually performed. The log can be sent to various destinations, including a database, and in several formats.

[29] It also allows the user to indicate which type of unrolling flavour to perform, whether to do loop peeling and what to do if the user gives an unroll or peel factor which is impossible for the loop and flavour. However, those capabilities, while useful for machine learning, would take a lot of space to describe without contributing much to the discussion.

[30] Strictly speaking, because of this extension point is built on top of `gcc-rtl-unroll-and-peel-loops.decision` join point, it can only guarantee to get the decision from the next advice down in the stack. Typically, however, this will just be the default heuristic.

[31] Match attributes where the pattern is a single * can be left out

A.0.6.5 Auto-Vectorisation

GCC has an optimisation to which tries to automatically vectorise loops. Vectorisation involves rewriting a loop to perform multiple iterations of a loop in one go. It is specifically supported by SIMD units, but might also be possible in some circumstances on the main CPU. The potential gains for successful vectorisation are large, but over zealous vectorisation can cause performance degradation.

libPlugin provides a plug-in to print information about vectorisation operations, to query various vectorisation aspects of loops and to control which loops are vectorised. It is a very similar plug-in to the loop unrolling plug-in. The essential difference is that rather than specifying the number of times the loop should be unrolled, the target is given. The target is unit that the code will be vectorised on (e.g. not at all, a SIMD unit, the main CPU, etc.)

A.0.6.6 Inlining

Function inlining replaces a function call site with the body of the function being called. Inlining is not always possible and not always desirable when it is possible.

There is a plug-in, `gcc-inlining`, in *libPlugin* which allows control of inlining. Again, the plug-in provides services to print information about inlining, query inlining characteristics (such as is a call site inlinable) and to control which call sites are inlined.

The most convenient way to control inlining is with the `gcc-inlining.override` extension point.

A.0.6.6.1 gcc-inlining.override This extension point allows inlining to be easily specified. The extensions consists of a list of wild carded `<call-site>` elements which match call sites within functions. When the compiler finds a call site it could inline it will scan the list of `<call-site>` elements until it finds a match and then use the inline value given in the element.

In the example below, all calls to function `bar` from function `foo` are inlined if possible; all calls to `blob` from `foo` will not be inlined and all other call sites in `foo` will use the default heuristic.

```
<extension point="gcc-inlining.override">
    <call-site main-input-file="foo.c" function="foo" callee="bar" inline="t
    <call-site main-input-file="foo.c" function="foo" callee="blob" inline="
    <call-site main-input-file="foo.c" function="foo" callee="*" inline="def
</extension>
```

This extension point does not distinguish between two different call sites in the same function to the same callee. Sometimes, this will require resorting to programming via the underlying join point.

A.0.6.7 Pass Manager

Passes[32] in GCC are where it performs the majority of its work. The compiler contains some 180 different passes which transform the code down closer to machine code and apply all of the numerous optimisations that GCC supports.

GCC contains a tree of passes; some passes are really just containers for others. Each pass has a unique identifier [33], although a pass may appear more than once in the tree; for example, common sub-expression elimination can be applied more than once. Each pass has a `gate` function which determines if the pass should be applied. Typically gates will check to see if suitable command line options have 'turned on' the pass. The pass also has an execute function which does the pass' work. Finally, there is a state machine [34] which should allow a developer to know ahead of time which passes can executed and to build a path through the passes however they wish, essentially dispensing with the default tree. This state machine is in a poor state of repair, however, since typical testing does not exercise it.

libPlugin comes with a plug-in which supports machine learning experiments at the pass re-ordering level as well as ordinary compiler extensions. The plug-in, `gcc-pass-manager`, offers facilities to add new passes to those already in GCC. It permits passes to be forcibly turned on or off, overriding their gate functions. It also allows complete control over the pass tree on a per function basis.

There is also a plug-in, `gcc-print-passes` which will log a list of all the passes each function is run through while being compiled. The log can be sent to various destinations, including a database, and in several formats.

[32] Other compilers may use the term 'phase' for GCC's 'pass'.

[33] In fact, in GCC 4.3 this is not the case, most passes are not named. The patch which adds *libPlugin* to GCC also ensures each pass has a unique name. In GCC 4.4 and above, passes are required to be named.

[34] The state machine is based on properties. An example property is "the function is in RTL form". The state of each function being compiled is described by nine properties which may either hold or not, giving 512 possible states for the function (in fact, not all property combinations are possible). Each pass lists the properties that must hold on a function for the pass to be able to execute on it. The pass also lists the properties that will become true or false as a result of running the pass on a function. With these it should be possible to plan a list of passes to apply which takes the function from the start state to the end state; i.e. to be suitable for sending to the assembler. This functionality would be very useful for machine learning on pass reordering and it is a shame that the state machine is out of date and incomplete.

A.0.6.8 Features

Machine learning experiments require features to be given to the model learner and, once a model has been created, to the model to get its predications. There are many different features that can be thought of for any code block. Researchers often have different needs, some will ask for features at the basic block level, some only on functions, others on instructions; and even if targeting the same level of code element, researchers will come up with different features.

libPlugin comes some ten or so plug-ins that compute features for different code elements. Most have been created to the specifications of other researchers. Two of the feature sets, Stephenson's and Milepost's, are well known having been reimplemented from other papers; they are described below. One of the more simple feature plug-ins, built for a colleague, is also described.

All the feature plug-ins have a very similar interface and similar capabilities. By default, all the plug-ins are lazy. If one of the feature plug-ins is loaded it will by default write out features in an XML format to a file called `<plug-in name>.xml` and will compute features for all functions it encounters.[35] However, for all feature sets, the output format, destination and a filter set of included functions can be configured by extending the plug-in's extension points.

A.0.6.8.1 Stephenson's Features StephensonStephenson and Amarasinghe (2005) showed that machine learning could successfully outperform the heuristics built by human experts. His experiments targeted loop unrolling and so he needed features capable of describing loops to the machine learning tools. *libPlugin* has an implementation of features in plug-in `gcc-features-loop-stephenson`.

The full list of Stephenson's features is given in table 6.10.

A.0.6.8.2 Milepost Features The MilepostFursin et al. (2008b) project produced a standard set of features given in table A.3. The features are produced by writing out the whole AST to a Datalog database and then each feature is represented as a Datalog relation over that database. The features are at the function level only.

libPlugin has a plug-in, `gcc-features-function-milepost` which reimplements the Milepost features. The implementation is simpler, just written in C, and considerably faster than the original Datalog implementation.

[35] Some feature sets operate on smaller elements than whole functions. The features are grouped by function, however.

Figure A.3: Milepost features

Number of basic blocks in the method
Number of basic blocks with a single successor
Number of basic blocks with two successors
Number of basic blocks with more then two successors
Number of basic blocks with a single predecessor
Number of basic blocks with two predecessors
Number of basic blocks with more then two predecessors
Number of basic blocks with a single predecessor and a single successor
Number of basic blocks with a single predecessor and two successors
Number of basic blocks with a two predecessors and one successor
Number of basic blocks with two successors and two predecessors
Number of basic blocks with more then two successors and more then two predecessors
Number of basic blocks with number of instructions less then 15
Number of basic blocks with number of instructions in the interval [15, 500]
Number of basic blocks with number of instructions greater then 500
Number of edges in the control flow graph
Number of critical edges in the control flow graph
Number of abnormal edges in the control flow graph
Number of direct calls in the method
Number of conditional branches in the method
Number of assignment instructions in the method
Number of binary integer operations in the method
Number of binary floating point operations in the method
Number of instructions in the method
Average of number of instructions in basic blocks
Average of number of phi-nodes at the beginning of a basic block
Average of arguments for a phi-node
Number of basic blocks with no phi nodes
Number of basic blocks with phi nodes in the interval [0, 3]
Number of basic blocks with more then 3 phi nodes
Number of basic block where total number of arguments for all phi-nodes is in greater then 5
Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
Number of switch instructions in the method
Number of unary operations in the method
Number of instruction that do pointer arithmetic in the method
Number of indirect references via pointers ("*" in C)
Number of times the address of a variables is taken ("&" in C)
Number of times the address of a function is taken ("&" in C)
Number of indirect calls (i.e. done via pointers) in the method
Number of assignment instructions with the left operand an integer constant in the method
Number of binary operations with one of the operands an integer constant in the method
Number of calls with pointers as arguments
Number of calls with the number of arguments is greater then 4
Number of calls that return a pointer
Number of calls that return an integer
Number of occurrences of integer constant zero
Number of occurrences of 32-bit integer constants
Number of occurrences of integer constant one
Number of occurrences of 64-bit integer constants
Number of references of a local variables in the method
Number of references (def/use) of static/extern variables in the method
Number of local variables referred in the method
Number of static/extern variables referred in the method
Number of local variables that are pointers in the method
Number of static/extern variables that are pointers in the method

A.0.6.8.3 Instruction Count Features The `gcc-features-basicblock-instructioncount` plug-in will create one feature for each type of AST or RTL node in each basic block or for each RTL.

A.0.6.8.4 Output Format *libPlugin*'s default feature plug-ins can all output the feature data in one of several formats. The current formats include XML, CSV, SQL and JSON. For example, if Stephenon's features were extended with the following format: [36]

```
<extension point="gcc-features-loop-stephenson">
    <output format="xml"/>
</extension>
```

[37] Then for each file compiled the features would print as XML like this:

```
<features type="stephenson" main-input-file="foo.c">
    <function name="bar" point="gcc-rtl-loop-init.execute.after">
        <loop number="1">
            <feature id="nesting.level" value="2"/>
            <feature id="num.ops" value="32"/>
            ...
        </loop>
    </function>
</features>
```

The output for other formats will be similar but if the format is not hierarchical (SQL and CSV) then the hierarchy is flattened. The hierarchy will also deeper or shallower depending on the granularity of the features set; for example, the Milepost features are function level, so the features are directly beneath the feature element and there are no loop elements.

The point at which the features were computed is described by the `point` attribute. This is discussed further in section A.0.6.8.8. In this case, the features are computed before the `rtl-loop-init` pass is executed, which is the default for Stephenson's features.

A.0.6.8.5 Output Destination It is often convenient record the features to a different file than the default. All of the feature plug-ins allow this with the `file` attribute

[36] For convenience, the output format and destination could be set by using command line arguments or environment variables through the plug-in variable syntax. This means that in the majority of cases, a plug-in does not need to be written.

[37] The extension point and the plug-in have the same name. Plug-ins and extension points inhabit different name spaces, so there is not conflict.

of the `<output>` element. The output format will be inferred if possible from the file name if the `format` attribute is not present.

In addition, the plug-ins all append the features to the file. This is convenient because benchmarks typically consist of more than one file and appending all the features means that the make file does not have to be altered. It is possible, however, to change this behaviour by using attribute `append="false"`.

```
<extension point="gcc-features-loop-stephenson">
    <output file="features.xml" append="false"/>
</extension>
```

Output can also be directed to a socket.

```
<extension point="gcc-features-loop-stephenson">
    <output host="server" port="12737"/>
</extension>
```

Multiple output destinations can be given and the plug-ins will send data to all of them.

A.0.6.8.6 Output to Database Recording data in a database is often more convenient than writing the data to file. *libPlugin* makes this automatic, relieving the researcher from the burden of doing it himself. Most of plug-ins that come bundled with GCC that report information about the compilation can direct their data to files, sockets or databases and the features plug-ins are no exception.

At present, only one database is supported, CouchDB. This database is document oriented meaning that data is loosely structured (in JSON) and is not relational. This style is excellent for researchers since it allows databases to quickly created and used without all of the headache associated with designing SQL tables and the difficulties that arise if the SQL layouts ever need to change. More databases may be supported at a later date.

To use the database, the plug-in must give server name or IP address, optionally a port, user name and password. The documents typically also need to be associated with some point in the iterative compilation space (otherwise the features will be meaningless). This can be done by giving `tag` attribute which will cause a field of the same name to be inserted into the document before sending it to the database. [38]

[38] There are also ways to add arbitrary JSON content into the document. This can be augmented with the `message` plug-in so that different information can be added for each function.

```
<extension point="gcc-features-loop-stephenson">
    <output
        db="couchdb" host="server" port="5984"
        user="hleather" pass="mypassword"
        tag="adpcm-compilation-1"/>
</extension>
```

A.0.6.8.7 Filtering functions It may be that a researcher will not be interested in features from all functions in all benchmarks. The features plug-ins can all specify a set of functions to include. The user gives wild-carded[39] lists of functions to get features for. Each element of the list either includes or excludes the functions it names and the elements processed in document order. A typical example might look like:

```
<extension point="gcc-features-loop-stephenson">
    <!-- Initially, all functions are included, but
     if we only want features for a few functions,
     then we start by excluding everything -->
    <exclude main-input-file="*" function="*"/>
    <!-- Now list what to include, two from foo.c
     and all in bar.c -->
    <include main-input-file="foo.c" function="foo"/>
    <include main-input-file="foo.c" function="bar"/>
    <include main-input-file="bar.c" function="*"/>
</extension>
```

A.0.6.8.8 When to Compute Features Features can often be computed at different points in the compilation. *libPlugin* allows features to be computed in response to any event. The `gcc-pass-manager` plug-in contains an event to mark the beginning and end of the execution of every pass and these are the typical events to which feature generation is attached although other events may be used. Each feature plug-in has a default event it will attach to if none is given. It may not be safe to attach to all events, for example, the function to get features for may not have be in the requisite state. At present, *libPlugin* does not sanity check the user's choice of event to see if it is suitable.

To choose the event to print features for, the user includes a `<when>` element with the identifier of the event's extension point. Multiple events can be given in each extension. [40] If different `tag` attributes or other data should be sent to the database on

[39] The wild cards used are POSIX glob patterns

[40] Passes can appear more than once in the pass manager's pass tree. However, there will be only one

each event, then different extensions should be used. The `point` will be described in the features. In the following example, the simple basic block instruction histogram features are generated before and after loop unrolling:

```
<extension point="gcc-features-basicblock-instructioncount">
    <when point="gcc-rtl-unroll-and-peel-loops.execute.before"/>
    <when point="gcc-rtl-unroll-and-peel-loops.execute.after"/>
</when>
```

A.0.6.8.9 API All feature plug-ins have an API so that they can be used from other plug-ins. For instance, once a machine learning model has been built, it should be reinserted into the compiler. The model will only be able to make predictions on new benchmarks by seeing the features for those benchmarks. The API allows this to happen relatively easily.

A.0.6.9 Data Dump

libPlugin allows full AST or RTL data of functions to be dumped to files. This is useful in situations where the researcher is unsure about what feature should be used, when debugging features or when using automatic feature generation.

The plug-in, `gcc-dump-ast-or-rtl`, prints AST or RTL of functions according to the state the current function is in when the dump is called. The dump is sent only to files, not a database simply because of the volume of data produced. The dump is configured in much the same ways as the features plug-ins; the output file can be specified and events that cause the dump can be given, too, with the same syntax.

---

event for the pass which will be fired multiple times per function if the pass is duplicated in the tree. This can mean that it is difficult to separate the features and to identify when the features were really generated. This problem will be investigated in the future.

Bibliography

Agakov, F., Bonilla, E., J.Cavazos, B.Franke, Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., and Williams, C. (2006). Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA. IEEE Computer Society.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aiolli, F., Da, G., Martino, S., and Sperduti, R. (2007). Efficient kernel-based learning for trees. In *Proceeding of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM).*

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. (2004). Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA. ACM.

Azad, R. M. A. (2003). *A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System.* PhD thesis, Ireland.

Back, T. (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, Oxford, UK.

Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. (1991). A static performance estimator to guide data partitioning decisions. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–223, New York, NY, USA. ACM.

Barricelli, N. A. (1957). Symbiogenetic evolution processes realized by artificial methods. *Methodos*, 9:35–36.

Beaty, S. J. (1991). Genetic algorithms and instruction scheduling. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 206–211, New York, NY, USA. ACM Press.

Bensusan, H. and Kuscu, I. (1996). Constructive induction using genetic programming. In Fogarty, T. and Venturini, G., editors, *ICML'96, Evolutionary computing and Machine Learning Workshop*.

Bernstein, D., Golumbic, M., Mansour, y., Pinter, R., Goldin, D., Krawczyk, H., and Nahshon, I. (1989). Spill code minimization techniques for optimizing compliers. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA. ACM.

Blackburn, S. M., McKinley, K., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA 2006, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*.

Bland, J. M. and Altman, D. G. (1996). Transforming data. *BMJ (Clinical research ed.)*, 312.

B.L.Welch (1947). The generalization of 'student's' problem when several different population varlances are involved. *Biometrika*, 34:28–35.

Bodin, F., Kisuk, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Prole 14 and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Bordes, A., Ertekin, S., Weston, J., and Bottou, L. (2005). Fast kernel classifiers with online and active learning. *Journal Of Machine Learning Research*, 6:1579–1619.

Box, G. E. P. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252.

Bradford, J., Fortes, J., and Bradford, J. (1999). Characterization and parallelization of decision tree induction.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1996). Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19.

Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M. F. P., Fursin, G., and Temam, O. (2006a). Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34, New York, NY, USA. ACM Press.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., P, M. F., and Temam, O. O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *In Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO*, pages 185–197.

Cavazos, J. and Moss, J. E. B. (2004). Inducing heuristics to decide whether to schedule. *SIGPLAN Not.*, 39(6):183–194.

Cavazos, J., Moss, J. E. B., and O'Boyle, M. F. (2006b). Hybrid optimizations: Which optimization algorithm to use? *Compiler Construction.*

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.*, 41(10):229–240.

Cavazos, J. and OBoyle, M. F. (2005). Automatic tuning of inlining heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14. IEEE Computer Society.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2005). Acme: adaptive compilation made efficient. *SIGPLAN Not.*, 40(7):69–77.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9.

Cooper, K. D., Subramanian, D., and Torczon, L. (2002). Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23:2002.

Dubach, C., Cavazos, J., Franke, B., Agakov, F., Fursin, G., O'Boyle, M. F. P., and Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, New York, NY, USA. ACM Press.

Dubach, C., Jones, T. M., Bonilla, E. V., Fursin, G., and O'Boyle, M. F. (2009). Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*.

Dubach, C., Jones, T. M., and O'Boyle, M. F. (2008). Exploring and predicting the architecture/optimising compiler co-design space. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 31–40, New York, NY, USA. ACM.

E.C.Fieller (1954). Some problems in interval estimation. *Journal of the Royal Statistical Society*, 16:175–185.

Epshteyn, A., Garzaran, M., Dejong, G., Padua, D., Ren, G., Li, X., Yotov, K., and Pingali, K. (2005). Analytic models and empirical search: A hybrid approach to code optimization. In *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC*.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and OBoyle, M. (2008a). Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC DevelopersSummit*.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., and O'Boyle, M. (2008b). Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005). Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA. IEEE Computer Society.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP*. SpringerVerlag.

Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324.

Koza, J. R. (1990a). Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer, J.-A. and Wilson, S. W., editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, 24-28, September 1990*, pages 366–375, Paris, France. MIT Press.

Koza, J. R. (1990b). The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In Soucek, B. and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York.

Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., and Jones, D. (2004). Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA. ACM.

Leather, H., Bonilla, E., and O'Boyle, M. (2009a). Automatic feature generation for machine learning based optimizing compilation. In *CGO '09: Proceedings of the International Symposium on Code Generation and Optimization*.

Leather, H., O'Boyle, M., and Worton, B. (2009b). Raced profiles: Efficient selection of competing compiler optimizations. In *LCTES' 09: Proceedings of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems*.

Lokuciejewski, P., Stolpe, M., Morik, K., and Marwedel, P. (2010). Automatic selection of machine learning models for wcet-aware compiler heuristic generation. In *Proceedings of the Fourth Workshop on Statistical and Machine Learning Approaches to ARchitectures and CompilaTion (SMART'10)*, pages 3–17.

Long, S., O'Boyle, M., and based Learning, U. I. (2004). Adaptive java optimisation using instance-based learning.

M.A.Creasy (1956). Confidence limits for the gradient in the linear functional relationship. *Journal of the Royal Statistical Society*, 18:64–69.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60.

Maoa, F. and Shen, X. (2009). Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proceedings of the International Symposium on In Code Generation and Optimization, 2009. CGO 2009*, pages 92–101.

Maron, O. and Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in neural information processing systems 6*, pages 59–66. Morgan Kaufmann.

Maron, O. and Moore, A. W. (1997). The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11:193–225.

Massalin, H. (1987). Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22(10):122–126.

Mcgovern, A. and Moss, E. (1998). Scheduling straight-line code using reinforcement learning and rollouts. In *In Proceedings of Neural Information Processing Symposium*. MIT Press.

Mierswa, I. (2004). Automatic feature extraction from large time series.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics.

Moss, E., Utgoff, P., Cavazos, J., Precup, D., Stefanović, D., Brodley, C., and Scheeff", D. (1998). Learning to schedule straight-line code. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems.*

Nisbet, A. (1998). Gaps: Genetic algorithm optimised parallelisation. In *In Proc. Workshop on Profile and Feedback Directed Compilation.*

O'Neill, M. and Ryan, C. (1999). Automatic generation of caching algorithms. In *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134. John Wiley and Sons.

P., A. and M.J.R., H. (1957). Interpretation of χ^2 tests. *Biometrics*, 13:113–115.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Paterson, N., Livesey, M., and Ss, K. (1997). Evolving caching algorithms in c by genetic programming. In *In Genetic Programming*, pages 262–267. MIT Press.

Pschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B. W., Xiong, J., Franchetti, F., Gai, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). Spiral: Code generation for dsp transforms.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning.* Morgan Kaufmann, San Francisco, CA, USA.

Ritthoff, O., Klinkenberg, R., Fischer, S., and Mierswa, I. (2002). A hybrid approach to feature selection and generation using an evolutionary algorithm.

Ryan, C., Azad, A., Sheahan, A., and O'Neill, M. (2002). No coercion and no prohibition, A position independent encoding scheme for evolutionary algorithms—the Chorus system. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278, pages 131–141, Kinsale, Ireland. Springer-Verlag.

Ryan, C., Collins, J. J., and O Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris. Springer-Verlag.

Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2:110–114.

Schlkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, USA.

Spector, L., Spector, E. L., Goodman, E., Wu, A., Langdon, W. B., m. Voigt, H., Gen, M., Sen, S., and Push, A. E. (2001). Autoconstructive evolution: Push, pushgp, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, 137146*, pages 137–146. Morgan Kaufmann Publishers.

Stephenson, M. and Amarasinghe, S. (2005). Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA. IEEE Computer Society.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003a). Meta optimization: Improving compiler heuristics with machine learning.

Stephenson, M., O'Reilly, U.-M., Martin, M. C., and Amarasinghe, S. (2003b). Genetic programming applied to compiler heuristic optimization.

Thomson, J., OBoyle, M., Fursin, G., and Franke, B. (2010). Reducing training time in a one-shot machine learning-based compiler. In Gao, G., Pollock, L., Cavazos, J., and Li, X., editors, *Languages and Compilers for Parallel Computing*, volume 5898, pages 399–407. Springer Berlin / Heidelberg.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.*, 44(6):177–187.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. (2003). Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, pages 204–215. IEEE Computer Society.

Vafaie, H. and DeJong, K. (1993). Robust feature selection algorithms. *Proc. 5th Intl. Conf. on Tools with Artifical Intelligence*, pages 356–363.

Vuduc, R., Demmel, J. W., and Bilmes, J. A. (2004). Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94.

Wald, A. (1947). *Sequential Analysis.*

Walsh, P. and Ryan, C. (1995). Automatic conversion of programs from serial to parallel using genetic programming – the Paragen system. In D'Hollander, E. H., Joubert, G. R., Peters, F. J., and Trystram, D., editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11, pages 415–422, Amsterdam. Elsevier, North-Holland.

Wellek, S. (2003). *Testing Statistical Hypotheses of Equivalence.* CRC Press.

Wetherill, G. and Glazebrook, K. (1986). *Sequential methods in statistics. 3rd ed.* London: Chapman and Hall.

Whaley, R. C. and Dongarra, J. J. (1998). Automatically tuned linear algebra software. In *CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING*, pages 1–27. IEEE Computer Society.

Whitehead, J. (1992). *The Design and Analysis of Sequential Trials.* Ellis Horwood.

W.J.Westlake (1972). Use of confidence intervals in analysis of comparative bioavailability trials. *Journal of Pharmaceutical Science*, 61:1340–1341.

W.S.Gosset (1908). The probable error of a mean. *Biometrika*, 6:1–25.

Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. (2003). A comparison of empirical and model-driven

optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76, New York, NY, USA. ACM.